NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

RE-PURPOSING COMMERCIAL ENTERTAINMENT SOFTWARE FOR MILITARY USE

By

Jeffrey D. DeBrine
Donald E. Morrow

September 2000

Thesis Advisor:	Michael Capps
Co-Advisor:	Michael Zyda

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2000	Master's Thesis

4. TITLE AND SUBTITLE
Re-Purposing Commercial Entertainment Software for Military Use

5. FUNDING NUMBERS
MIPROEMANPGS00

6. AUTHOR(S)
DeBrine, Jeffrey D. and Morrow, Donald E.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Economic & Manpower Analysis
607 Cullum Rd, Floor 1B, Rm B109, West Point, NY 10996-1798

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Virtual environments have achieved widespread use in the military in applications such as theater planning, training, and architectural walkthroughs. These applications are generally expensive and inflexible in design and implementation. Re-purposing these applications to meet the dynamic modeling and simulation needs of the military can be awkward or impossible.

Video games are designed to be both technologically advanced and flexible in design. We evaluated current games and modified Quake 3 Arena™ (Q3A) to serve as both an architectural walkthrough and a primitive team trainer. To accomplish this, we incorporated a real Naval Postgraduate School building into Q3A. We also modified the game's source code, characters and their behaviors, weapons models and characteristics, and overall gameplay.

By re-purposing commercial entertainment software, we have produced a viable military virtual environment application that is less expensive yet arguably as engaging as current computer-based options. This application was created in approximately 300 man-hours with a cost of $6780 (including hardware)—far less than the development time and cost of similar military virtual environment applications. Game evaluations included in this thesis facilitate and inform similar modification efforts by highlighting entertainment technology available in the year 2000 game market.

14. SUBJECT TERMS: Modeling and Simulation, Software Re-Purposing, Video Games, Entertainment Technology, Architectural Walkthrough, Game Modification

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFI- CATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239- 18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

RE-PURPOSING COMMERCIAL ENTERTAINMENT SOFTWARE FOR MILITARY USE

Jeffrey D. DeBrine
Lieutenant, United States Navy
B.S., United States Naval Academy, 1992

Donald E. Morrow
Lieutenant, United States Navy
B.S., Auburn University, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN
MODELING, VIRTUAL ENVIRONMENTS AND SIMULATION

from the

NAVAL POSTGRADUATE SCHOOL
September 2000

Authors: _____
Jeffrey D. DeBrine

_____
Donald E. Morrow

Approved by: _____
Michael Capps, Thesis Advisor

_____
Michael Zyda, Co-Advisor

_____
Rudy Darken, Academic Associate
Modeling, Virtual Environments and Simulation Academic Group

_____
Michael Zyda, Chair
Modeling, Virtual Environments and Simulation Academic Group

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Virtual environments have achieved widespread use in the military in applications such as theater planning, training, and architectural walkthroughs. These applications are generally expensive and inflexible in design and implementation. Re-purposing these applications to meet the dynamic modeling and simulation needs of the military can be awkward or impossible.

Video games are designed to be both technologically advanced and flexible in design. We evaluated current games and modified Quake 3 Arena™ (Q3A) to serve as both an architectural walkthrough and a primitive team trainer. To accomplish this, we incorporated a real Naval Postgraduate School building into Q3A. We also modified the game's source code, characters and their behaviors, weapons models and characteristics, and overall gameplay.

By re-purposing commercial entertainment software, we have produced a viable military virtual environment application that is less expensive yet arguably as engaging as current computer-based options. This application was created in approximately 300 man-hours with a cost of $6780 (including hardware)—far less than the development time and cost of similar military virtual environment applications. Game evaluations included in this thesis facilitate and inform similar modification efforts by highlighting entertainment technology available in the year 2000 game market.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

3D	Three Dimensional
AI	Artificial Intelligence
BOT	Computer-controlled character (i.e., robot)
BSP	Binary Space Partition
CAD	Computer-Aided Design
COTS	Commercial, Off-the-Shelf (used to describe software)
CTF	Capture the Flag
FPS	First Person Shooter
MMP	Massively Multiplayer
MOD	Game Modification
NDL	Numerical Design Limited
PC	Personal Computer
PvP	Player Versus Player
PVS	Potentially Visible Set
Q3A	Quake III Arena
RAM	Random Access Memory
RPG	Role Playing Game
SKIN	Player's Appearance (stored as a texture)
TD	Team Deathmatch
UT	Unreal Tournament
UI	User Interface
VE	Virtual Environment
VR	Virtual Reality

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENT

The authors would like to acknowledge the financial support of Dr. Casey Wardynski at the Office of Economic & Manpower Analysis for enabling the purchase of equipment and software used in this thesis.

The authors would also like to thank Dr. Michael Capps for his direction and enthusiasm for this thesis. The entire NPSNET research group has been essential in the successful completion of this thesis. Additionally, David Falby was very helpful with his game evaluation assistance.

Lastly, we would like to thank our families for their support and understanding: LT DeBrine would like to thank Ami for her love and tolerance during this seemingly endless task, along with his two children for support. LT Morrow would like to thank Suzanne for her love, support and assistance during these last few months.

THIS PAGE INTENTIONALLY LEFT BLANK

I. OVERVIEW

A. THESIS STATEMENT

By re-purposing commercial off-the-shelf (*COTS*) entertainment software, it is possible to produce viable military virtual environment applications that are less expensive yet more engaging than current computer based options.

B. MOTIVATION

Virtual environment and virtual reality applications are achieving widespread use in the military. Military VE applications include theater planning, training, and architectural walkthroughs. The primary benefit of VE applications is that they provide an environment that is safe and can simulate virtually any real-world situation at any time. These applications are currently relatively expensive and inflexible in that a single computer is often dedicated to one type of application, and that very application is often developed specifically for that single task.

1. Military VE Training Applications

There are several types of computer-based VE trainers. At the simplest level are single-user applications where the other members of the team in the trainer are computer-controlled agents. A good example of this type of trainer would be a flight simulator where the other planes in the pilot's formation are computer controlled. The second level brings interactivity between team members. These types of trainers utilize networked applications where each member of the team is presented with its own view of the environment (for example, the same flight simulator, except that the other planes in the

formation are controlled by other members of the team). The highest level of computer-based team trainers today are full-scale mockups of an environment where the users may not even realize that they are interacting with a computer at all, such as a Combat Information Center Team Trainer Mockup.

These trainers are very specialized and are often developed for one specific mission or task. As a result, their development can be expensive and time consuming. Usually there is no mechanism to modify the trainer to reflect different changes or desires, other than the "canned" changes that have been programmed into the system. For example, a popular computer-based team trainer is the mock-up of the control room for a nuclear power plant. The computers that control the simulations have been pre-programmed with certain casualties. Introducing a new casualty into the system requires reprogramming the computers. Changing the physical layout of the mock-up to reflect changes to the real-world control room is potentially even more difficult. These types of trainers, while they offer enhanced realism, are usually impossible to modify once they have been constructed.

2. Architectural Walkthroughs

Architectural walkthroughs of virtual environments have been popular applications in the VE/VR field for several years. These applications help users spatially visualize an environment by presenting a three-dimensional (*3D*) representation of the environment. These representations allow the user to navigate around in that environment. The military, along with civilian industry, uses 3D architectural walkthroughs when designing and prototyping buildings, vehicles, and other real-world environments to help determine their habitability. Two of the more famous architectural

walkthrough projects are the University of North Carolina, Chapel Hill and the University of California, Berkeley Walkthrough Projects (Singhal & Zyda, 1999).

Architectural walkthrough applications are usually specifically programmed for the individual task, or are done with computer-aided design (*CAD*) software. Dedicated architectural walkthrough applications are very expensive and slow to develop, but have good visual fidelity. CAD software such as AutoCAD® is also very expensive, and generally has poorer visual fidelity than dedicated applications because these applications are primarily for design, not for visualization.

3. COTS Entertainment Software

COTS entertainment software has the capability to overcome many of these limitations if properly modified. These products are designed to be modified relatively quickly and cheaply. Therefore, today's cutting-edge networked and graphics enhanced games can easily produce high fidelity, compelling simulations of real-world scenarios. These simulations can serve as trainers, architectural walkthroughs, or theater planners. More information on defense and entertainment collaboration can be found in a report by the National Research Council: *Modeling and Simulation: Linking Entertainment and Defense* (1997).

C. APPROACH

It was our goal to determine the feasibility of reutilization of COTS entertainment software for military VE training and architectural walkthroughs and to evaluate the difficulty in doing so. We decided that a mock team trainer would be beneficial as a proof-of-concept application. We therefore set out to determine how difficult it would be

to modify an existing game into a proof-of-concept mock application with team-training and architectural walkthrough characteristics. We did not attempt to implement any training metrics into our proof-of-concept; we merely meant to see if such application could be built.

We first evaluated existing entertainment software to see if there were any products that would immediately meet our needs. The results of this analysis are discussed in Chapter III of this thesis. Because no COTS entertainment existed that would directly support our requirements, we tried a re-purposing effort.

We decided to modify Quake 3: Arena® (*Q3A*) to see if it would meet our needs. Our plan was to devise a scenario that would require the players to designate a leader, develop a plan, disseminate the plan to the other players, and execute the plan. We chose a hostage scenario that would take place in Herrmann Hall. To accomplish this, we first created and imported a model of Herrmann Hall into Q3A. We then modified one of the computer-controlled characters (*bot*) so that he would behave passively and not attack other players—that is, be the hostage. We also modified one of the weapons so that it could be used as a laser pointer. A leader could then point at a door with the laser pointer and instruct another player to guard or perhaps enter through that door. We also modified the gameplay so that each player would have only one "life", and not automatically regenerate after being killed. This application will show the degree of difficulty of modifying an existing game to suit military purposes.

D. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

Chapter II: Background. Chapter II provides the reader with a background in the technical areas of this thesis. It also includes an analysis of previous related work in this field.

Chapter III: Current Game Technology. This chapter provides an in-depth analysis of current COTS entertainment software, as it pertains to the areas of this thesis—namely its capability to be modified and serve as our proof-of-concept application.

Chapter IV: Proof of Concept. Chapter IV provides a detailed description of our decision process and a general description of the steps required to build our proof-of-concept application.

Chapter V: Implementation. Chapter V provides an in-depth description of the method to create our proof of concept.

Chapter VI: Analysis. This chapter discusses the benefits that we feel our system has over other options, possible future thesis work in this area, and final conclusions and recommendations.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND & RELATED WORK

A. INTRODUCTION

This chapter offers the reader sufficient background to appreciate the methodology of this work. The first section is devoted to game technology and terminology. Previous attempts to re-purpose commercially available games are also covered. The basics of team interaction and how team members differ from team leaders are highlighted in the final section.

B. INTRODUCTION TO GAME TECHNOLOGY

Game technology has developed its own terms and language to describe the parts that make up a commercially developed game. For the reader's convenience, this section describes both the program parts that manufacturers provide to purchasers and the myriad code changes that programmers are able to add to these games. These changes can alter the original content of the game or sometimes completely change the game itself. Sometimes programmers just want to change a weapon or level in a game while keeping the basics of the game intact. Others may want to take the focus of a game and change it into a completely different program that still uses the code that the game was built on. These differences are both accommodated with current game technology. Figure 1 shows the varying complexity of modifying a video game.

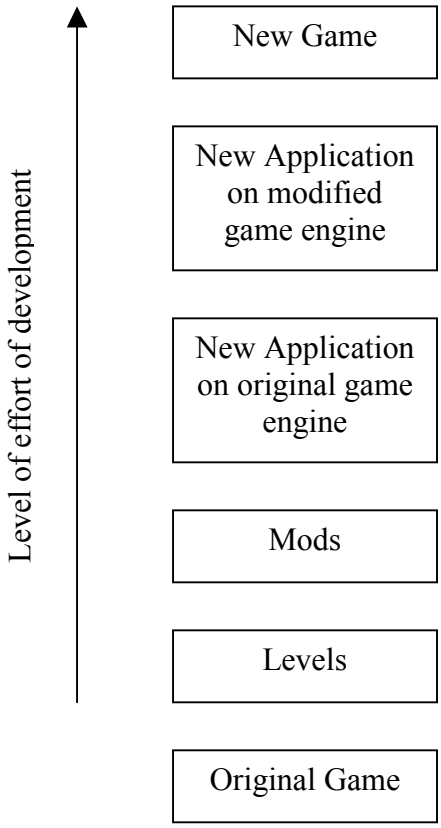

Figure 1. Level of Effort to Modify an Existing Game

1. Hardware Platform

There are many different platforms available for games to be played on. The most prevalent is the personal computer (*PC*). This platform is useful because it is multi-purpose, and can be configured to contain hardware that is needed for playing commercial games. Because of the high processor speed and large amounts of random access memory (*RAM*) used in today's personal computers, they can be as capable as game-specific platforms. One example of a game-specific platform is the Sony PlayStation®. This hardware device is built specifically to play games using a separate video display. Since this platform is only used for games, it has superior graphics and

fast game-play. The Sega DreamCast™ and Nintendo 64™ are two competitors with the PlayStation that include the same basic features. In 2000, the Sony PlayStation® 2 raised the bar in the game hardware arena, with the fastest graphics and processing to-date in a game-only platform. Part of the tradeoff with these types of game-specific platforms is the difficulty of programming new games for them. They require specialized development and production in order to produce compatible games.

2. Game Engine

Each commercially available game is built upon a game engine. This is the part of the game that is independent of the story and scenarios presented during game-play. The engine is responsible for the graphics, visibility calculations, artificial intelligence of enemies, network communications, etc. The game engine is generally not modifiable and is usually built completely separate from the application code. This provides a layer of abstraction that allows for engine reuse in different applications. Some engines are sold independently, allowing the developer to build a new game application atop the reusable engine. One example of this kind of game engine is the NetImmerse application from Numerical Designs Ltd. Prince of Persia™ 3D is a commercial game that was built on the NetImmerse engine. In games such as Quake™ III and Unreal™, the engine was developed with a specific game in mind. Still other game engines are developed so that they may be used in conjunction with other engines. These partial engines provide a certain function, such as physics calculations, in order to speed the development of the larger engine.

3. Application

The game plays on top of the engine. The application part of the code provides the user with the scenarios that can be played and the enemies that the user will play against. The application code contains scripts for character control and maps for the levels in the game. It also includes the textures and skin coverings (called *skins*) for the animated figures representing the players in the game (called *avatars*), along with all of the other models, including guns, enemy avatars, etc.

4. Modifications

Many games release their source code to the public in order to allow users to modify the game's playability. In today's market, games must always be seen as "cutting edge" technology with fresh stories and scenarios. This release of the code to the public allows users to change the code, thereby keeping the game up to date, without additional effort by the game development company. Players must still buy the game in order to run the modified code. These modifications, which are called *mods*, are available for free over the Internet. The companies may not make any money off of mods directly, but these changes allow the game to continue to evolve even after the final version is released by the parent company. This continual evolution of a game extends the life of the game beyond that of the original application. Examples of games that can be freely modified by any willing programmer are Quake I, II and III and DOOM.

The use of mods has become quite extensive in recent years. While companies release their source code and allow it to be modified, the code must still run on the game engine. This ensures that people will still buy the original game product, thereby extending shelf life. There are numerous websites devoted to making, distributing,

debugging and designing new mods. Owners of these modifiable games can go online, download a new mod and be playing a new game with the same engine in just minutes.

C. PREVIOUS MODIFICATIONS TO COTS SOFTWARE

The ability to modify available game application code or game scenarios has been used to make COTS game software useful for practical non-gaming applications. Many researchers and businesses have found it easier to modify existing code to fulfill their needs rather than start from scratch. Also, using a commercial game as a starting point has allowed researchers to use cutting-edge graphics techniques that could take years to develop on their own.

1. DOOM Trainer

The United States Marine Corps wanted to develop an application that would allow soldiers to train in small groups. The programmers took the engine from DOOM, one of the original first-person shooter (*FPS*) games. This trainer allowed up to four people to train simultaneously at separate computers. The soldiers all saw the same map, but from their own perspective, and could communicate with each other via headsets. The main purpose was to train for squad interaction by having the soldiers talk to each other as they proceeded through a stressful combat scenario. No attempt was made to measure team interaction in the trainer.

2. Multi-User Information Visualization

The Computer Science department at University of Durham in England created a project that allowed multiple users to visually explore program code. The reasoning behind the research was that people have a hard time understanding code that they did not

write (Knight & Munro, 1998). They hypothesized that it might be possible to exploit human navigation capability in understanding program architecture if the code could be explored in a 3-dimensional fashion. The researchers decided to modify existing game technology due to the low cost of obtaining the game engine and the ease in which they could get a 3-dimensional map up and running.

This project was implemented on the Quake 2 game engine. Rooms in the 3-dimensional world represented functions in the code. If a room was connected to another via a hallway, then those two functions were somehow connected in the code. Researchers also added other models to aid users during the walkthrough. Torches were placed to help highlight areas of interest and laser pointers were available to show other users where to go. At the publishing of the technical report, only very simple code had been implemented in the walkthrough. They found that rendering complex code in 3 dimensions would be difficult, if not impossible. However, they were able to show that modified COTS software could be a powerful tool for creating walkthroughs for nearly any purpose (Knight & Munro, 1998).

3. Unrealty

The commercial real estate industry has been searching for an easy way to produce a walkthrough of a large structure in 3 dimensions. This would allow sellers to show customers many different available buildings without ever leaving the office (Miliano, 1999). The problem has been that large building walkthroughs are very difficult produce from scratch and are generally slow and unpleasant to use with a slow frame-rate. To meet this need, Unrealty was designed to provide the real estate industry with the ability to create architectural walkthroughs for buildings for sale. They used the

game engine from Unreal™ Tournament, a FPS game similar to Quake™. Unreal™ Tournament is known for rich texturing, fast rendering, and easy modification. The Unreal™ Tournament game engine is able to render up to 60,000 polygons per building (which is more than adequate detail for most structures) in real-time.

4. Land Warrior

NovaLogic, in conjunction with Training and Doctrine Command Analysis Center (TRAC) Monterey, is producing a modification to the Delta Force 2 game called Land Warrior. This application will be used to enhance training with experimental Land Warrior equipment. A new user interface was added to the game to incorporate Land Warrior devices and communication techniques. This interface simulates the monocle that soldiers wear, allowing the user to cycle through myriad display options such as maps, scope view, etc. More information can be obtained at the Land Warrior official website: http://www.novalogic.com/

D. TEAM DYNAMICS

In order for us to understand the complexities of incorporating teamwork into a game, we had to first examine the qualities that make a good team leader or team member. There are specific qualities that are desired of group members that are different from the qualities expected in a group leader. Even though our final product is a proof of concept that does not actually measure team interaction during gameplay, we needed to learn these qualities so we could determine whether they might be implemented into a game.

1. Characteristics of Effective Group Members

First we must look at the desired characteristics of a group member, or follower. A good follower conforms to group decisions; McGraw and Bearden (1990) showed that those who were able to conform to the group were able to best fit into the military lifestyle. Those that chose the path of independence most often sought a discharge from the military. This trait could easily be measured in the game scenario just by observing how the individual performs in the group. A good design could measure whether the player stays with the group or detaches on a self-proclaimed mission. The game could also measure how well the player follows the instructions of the leader during the scenario. Lastly, the game scenario could determine how effectively the player works with the team in order to achieve the team goal rather than seek to fulfill individual needs.

The second desired trait in a follower is that the individual is either very skilled at the task, or is very interested in, and committed to, a successful outcome (Foushee, 1984). Foushee states, "A group with individuals who are skilled at the task or very interested in doing well will likely perform differently than a group comprised of unskilled, uninterested members." The desire of the individuals that are participating in the game scenario could be determined via a questionnaire. Then the difference between teams with desire and those without could be determined. However, those that do not express a great desire for this canned scenario cannot immediately be disqualified as non-team oriented individuals. Desire should only help in this experiment, not serve as a detractor.

There are several factors that will decide whether a member of a team will be a successful member of that group. As Forsyth (1990) reports, a member must have a clearly defined role in the group. Also, according to Weiner (1990), the member must have a leader that the member feels is supportive of his/her efforts. Having a clearly defined role in the group is important in order to avoid inter-personal conflict within the group. With well-defined roles in place, the group is able to function more harmoniously, leading to greater individual satisfaction within the group (Forsyth, 1990).

2. Characteristics of Effective Leaders

A leader that is supportive of the followers in the group is often a greater motivator than peer support in the same group. A study found that the degree to which a follower believed his/her leader to be supportive was a greater influence in motivation performance than the perceived amount of support from peers (Weiner, 1990).

In order to measure the leadership ability of a team member, the factors that make a leader must be quantified. These leadership characteristics include:

- is supportive of his/her followers
- conforms less than his/her followers
- has relationship behaviors that improve relations between group members
- has task behaviors that aid the group in successful task completion
- leads with "transformational leadership"
- in this application, a good leader rules by legitimate power and expert power, not referent power

While each of these traits is important, some are impossible to measure in a game scenario. In fact, none of these traits can be measured automatically by an algorithm in

the game. More likely, they will have to be derived from input from the followers in the scenario. They will be the most affected by the leader's decisions and therefore most able to comment on the leader's effectiveness.

3. Summary

As of now, no one has been able to implement all of these characteristics in a game or simulation. At this time, there is no computer application that can automatically measure team dynamics. Once these team interactions can be implemented in a simulation, there is no reason that they cannot also be incorporated into a game modification. The stumbling block for measuring team dynamics is algorithmic, and not dependent on the application that is being run.

III. CURRENT GAME TECHNOLOGY EVALUATION

A. INTRODUCTION

The purpose of this chapter is to evaluate current games on their utility and ease for re-purposing and collaboration. Many games today already offer advanced teamplay that could potentially meet military needs without having to create a proof-of-concept. If no current game meets our needs, our next choice will be to determine the difficulty of modifying a game to meet those needs. Some of today's games have the capability to be modified, based on the user's desires, as discussed in Chapter II. This chapter summarizes the aspects of each game that can be modified by the user.

One other important aspect of current games is the fact that some of them are written using a third-party game engine. It is important to know which games use third party engines because it illustrates what other capabilities a particular engine has. For example, if a game utilizes an engine from another game, yet that new game exhibits different characteristics from the original game (the game from which the game engine was obtained), it is then possible to visualize that particular capability of that engine. This information can be very useful when deciding what game engine to use when faced with a re-purposing task.

The next section defines the terms and criteria we chose for evaluation. The following section summarizes and tabulates the results of these evaluations. This chapter then discusses the more popular game engines used by these games and closes with an analysis of our decision process for choosing a game to serve for our proof-of-concept.

The games chosen for evaluation were picked primarily for one of two reasons. First is popularity and market penetration, because more sold copies directly correlates with mod success. Secondly, some games were chosen for unique teamplay and cooperation characteristics. We tried to select products representative of most major games.

B. TEAMPLAY IN GAMES, DEFINITIONS

This section defines terms and criteria used in evaluating the games.

Networked play: Does the game support network play? If a game does not have networked gameplay such that one human player can interact with another human player over a local network or the Internet, then it will not likely offer a good teamwork skill-building environment. Of course, many non-networked games have bots that the human player can interact with, but no games today had accurately programmed the artificial intelligence to simulate the interactivity that usually takes place between live humans.

Cooperative gameplay: Does the game support cooperative gameplay among its human players? In other words, are the human players on the same team, or are they forced to play on opposing teams?

Teammates bots or humans: Are the human's teammates bots or humans (or both)?

Communication method: What types of communications are included in the game? Very few games today have actual voice communication between human players, and none evaluated directly support voice recognition. Most games have a means to enter text communications and commands to human and bot players. Some games only had provisions to enter "pre-canned" messages via "hot-keys" to other players. Games

such as this may have only a very few messages to choose from (less than six or so) or very many. The commands are sometimes context specific- that is, the communications to choose from change depending on the current situation. Finally, once a command is entered, that command may or may not actually alter the behavior of the player(s) at which the command was directed. For instance, if a command was given to a player to guard a certain door, if that player was then forced to stay within a certain distance of the door, then that player's behavior was modified. On the other hand, if nothing changes after that command has been given, and the player could easily ignore the command, then the commands did not alter the behavior.

Number of simultaneous online players: What is the maximum number of players that can play in a single game? This is different from the number of players that can connect to a given server. Some game servers can have many games running on them at the same time. Within each game, there are limitations to the number of players that can play at the same time. There are also limitations to the number of games that can simultaneously be played on a server, but this is very hardware dependent.

Concepts of team leaders and followers: Are all players viewed as equals in the chain-of-command, or are there concepts of a leadership hierarchy? Do the leaders lead humans, bots, or both? Is there more than one level of leadership?

Scoring and evaluation: How are individual players evaluated or scored? How are teams evaluated and scored? This is especially important for our re-purposing efforts. For instance, game where the number of kills a player has determines that player's score may not easily serve in navigation training scenarios.

Level of modification available: Many of today's games can be modified on different levels. Aspects of games that can be modified are maps, the player's appearance (skins) and skills (Artificial Intelligence), weapon's appearance and functionality, missions, and animations. Some game developers have released the source code for their games so that the actual gameplay and game logic may even be altered. Of course a modification of this type is very complex and requires knowledge of the programming language that the game was written in. All modifications require some type of knowledge about the tools and syntax used to create the modification. A game may allow modification of any or all of these features.

C. GAME EVALUATIONS

This section contains the evaluation and summaries of twenty current popular video games. There are three types of games evaluated here: role-playing-games, first-person-shooter type games, and one sport simulation game.

Role-playing-games, or *RPGs*, usually place many (often hundreds of) players in an environment where they can interact with each other. Often times, they have the ability to fight, cast spells and magic, buy and sell, and otherwise interact with other players. They often encourage the forming of teams – in which case the teams have the capability to accomplish more than an individual could accomplish. These types of games often discourage the killing of other human players. In fact, some of these types of games even have the option to make players invincible to other human player attacks.

First-person-shooters are defined by the first-person view that the player is presented with and the fact that the player shoots his or her weapon at opposing players. They are usually player-versus-player (*PvP*) type games that place the player against

everyone else in the game (free-for-all). They usually do little to encourage teamwork, but some games have made some advances here. There are few other types of games that present the player with a first-person view of the environment other than "shooter" type games.

Finally, there is one sport simulation game evaluated here. This type of game places the player into the sport and tries to recreate the environment of an actual game while competing against human or computer opponents.

1. EverQuest™

EverQuest is one of the most popular role-playing games of recent times. It is a massively multiplayer (*MMP*) game, capable of supporting hundreds or thousands of simultaneous players online. It is a lot like other role-playing games in that players choose a player class with specific characteristics. Players can form parties, decide on a team leader, and seek out objectives to complete as a team. Each individual on the team will then share the reward. The missions and objectives are complex enough such that a single player could never accomplish the missions by themselves, thereby requiring the teams to be constructed carefully, covering all necessities. EverQuest builds teamwork by requiring individuals to find a team that benefits from that individual's membership. Players can also join guilds. These guilds are less structured and do not tangibly benefit the player the same way that parties do.

The company that created EverQuest™ maintains servers to create a persistent environment for each player. The servers are not interconnected; therefore, each server maintains an independent environment, separate from every other server's environment.

No modifications can be made to EverQuest™. There are no level editors and the source code has not been released. The only changes that can be made are to the player's own character, using the character editor built in to the program.

EverQuest™ was written and developed by 989 Studios (Sony owned). Official web site: http://www.station.sony.com/everquest/

2. Asheron's Call™

Asheron's Call™ is a role-playing game that is played online. It is very similar to other RPGs evaluated here. It encourages interaction among players, and supports the formation of teams, or fellowships, in which players adventure together and share the experience. Asheron's Call™ is not normally a PvP game; however, there are servers where a player can connect that allow this type of playing. Most players choose to play socially on the other servers. Asheron's Call™ does not support any consumer modifications.

Asheron's Call™ was written and developed by Microsoft. Official web site: http://www.zone.com/asheronscall/start.asp

3. Ultima™ Online: The Second Age

Ultima™ Online is a MMP game (in fact, it was the first MMP game), like EverQuest™, and can support thousands of simultaneous online players. Ultima™ Online is also a role-playing game, where each player develops his or her character and interacts with the other character/players in the environment. Ultima™ Online also supports the formation of teams, like EverQuest™.

Ultima™ Online was written and developed by Origin. Official web site: http://www.uo.com/

4. Delta Force™ 2

Delta Force™ 2 is a FPS game that places the player into an elite squad of Special Forces commandos. This game was built on the Voxel Space game engine. The player can explore various missions as the squad leader. However, this does not mean that the player is actually in control of the other squad members. The computer is in charge of the movements of the other members of the team, although the player can control some of their movement. In multi-player mode, the player can join with other humans to play the game. No control is possible from one human to another: compliance is optional. One interesting feature is the Voice-Over-Net™ capability, allowing players to communicate with each other using voice rather than text. Also, Delta Force™ 2 comes with a mission editor that allows players to create their own missions.

NovaLogic hosts an online game, which can support up to 50 players simultaneously, or any player can host their own game. Multi-player games can take on various forms, such as cooperative, Capture the Flag (*CTF*), King of the Hill, and others. These games center on the concept of cooperative teamplay between humans in the same squad trying to achieve some objective. However, as previously stated, there is no requirement that team members follow any orders given to them. There is no measurement of team effectiveness, other than the number of kills and the time to complete the mission in cooperative scenarios. In the others, such as CTF, the number of objectives achieved is the score for the team.

Delta Force™ 2 was written and developed by Novalogic. Official web site: http://www.deltaforce2.com/

5. Half-Life®: Opposing Force™

Half-Life® was created using a heavily modified Quake II game engine. As a first person shooter type game, it is very similar to others of this type. However, Half-Life® is a very modifiable game. It was created with the intention of being modified, and is therefore rather easy to modify. Many exciting mods have already been created for Half-Life®. Two of the most interesting mods so far are Team Fortress and Counter-Strike, and both are reviewed in this chapter.

Half-Life® "as-is" does not have any teamplay concepts relevant to this study. Other than the ability to type text messages to other players, there is no provision to establish a lead-follow relationship among players.

Half-Life® was written and developed by Valve Software (owned by Sierra Studios). Official web site: http://www.sierrastudios.com/games/half-life/

6. Counter-Strike

Counter-Strike is an interesting modification of Half-Life. It places two teams, one terrorist team, and one counter-terrorist team against each other. The goals of the counter-terrorists include rescuing hostages and VIPs and defusing bombs, all while the terrorists are working to eliminate the counter-terrorists.

Counter-Strike encourages teamwork more than most games. Most maps are designed to have only two or three entrances into the building, which often forces the counter-terrorists to send small squads to each entrance. If a single player were to try and

get through an entrance, the waiting terrorists would almost certainly eliminate the player. There are no provisions for a leadership hierarchy. All players are considered equal in the chain of command.

Counter-Strike cannot be modified further, other than superficial changes such as maps and character skins. It is still being developed and the source code is not available.

Official web site: http://www.counter-strike.net/

7. Team Fortress Classic

Team Fortress, originally a modification of Quake II, but later a very popular modification of Half-Life, is another interesting teamwork-building game. This game is similar to other capture the flag type games, but the difference here is that there are different player classes, similar to the RPGs mentioned previously. Each player chooses a role (such as medic, mechanic, heavy weapons guy, or spy) that they will play at the beginning of each game. Different classes have different characteristics and capabilities. The more successful teams have a good balance of different classes. These successful teams also communicate well within the team and work together to capture the enemy's flag.

Like Counter-Strike, Team Fortress is still in development and cannot be modified further. There is a Team Fortress mod in progress for Quake III, which should be similar to the Team Fortress mod for Half-Life.

Team Fortress Classic was written and developed by Valve Software. Official web site: http://www.valvesoftware.com/projects.htm

8. Quake III Arena™

Quake III Arena™ (*Q3A*) was developed uses its own game engine, which was a improved version of their own Quake II engine. The Q3A engine (as well as the Quake II engine) is available for purchase by other parties. Q3A has networked play, and other human players can either play on the same teams or on opposing teams. Q3A offers two teamplay modes: CTF, and Team Deathmatch (*TD*). The basic premise behind both games is to kill the members of the opposing team.

Q3A does introduce the concept of team leaders and followers with some interesting concepts. A player can designate himself the leader and give pre-determined commands to other players. If an order is given to a bot, then that bot's behavior is modified to follow the order. If an order is given to a human player, there is no change in that player's behavior. However, any player can designate him/herself the leader and give orders. In fact, a player can give orders to anyone with even designating him/herself a leader at all. The bots always follow their last order, no matter who gives the order. Finally, there are only two levels of leadership – leader and follower, but with the exception described above.

Players and teams are evaluated separately. In CTF, team scores are kept by recording the number of times the enemy's flag is captured. Team scores in TD are kept by recording the number of kills each team has. Individual scores for both TD and CTF are kept by recording the number of kills each player has.

Q3A has been heavily modified since its initial release. Just about every aspect of the game can be modified, primarily through altering the source code and recompiling the game.

Quake III Arena™ was written and developed by id Software. Official web site: http://www.idsoftware.com

9. Tom Clancy's Rainbow Six

Rainbow Six is a game in the Special Forces genre that places the player in the role of leader of a group of anti-terrorists. This game is unique in the fact that it requires a great deal of planning before any mission can be attempted. The user is required to choose each member of the team, choose all weapons that will be used, and develop the plan of attack, as well as determine the rules of engagement before getting to the meat of the missions. Since the player has the responsibility of creating a mission plan, the computer-controlled team members adhere to this plan during the missions, rather than waiting for the leader to issue commands on the fly.

The multiplayer aspect of this game allows different players to play a game over the Internet. In cooperative mode, players can fill any of the roles of the team and then proceed to a mission scenario. This cooperation is strictly voluntary, since no commands that are given are mandatory. The computer can fill in any vacancies with computer-controlled team members if there are not enough humans available. The other available mode is adversarial, where players can join either the blue or gold team and then play a mission as enemies.

Rainbow Six was written and developed by Red Storm Entertainment. Official web site: http://www.tomclancy.com/rainbow_six/

10. Tom Clancy's Rainbow Six Rogue Spear™

Tom Clancy's Rainbow Six Rogue Spear™ is Tom Clancy's Rainbow Six with 18 added missions. No functionality significant to this study was added.

11. Soldier of Fortune®

Soldier of Fortune® is a classic FPS game, built on the Quake II game engine, with single- and multi-player scenarios. The single-player game places the user in a secret agency that is trying to rid the world of terrorism. The player then tries to kill everyone on the screen as the game proceeds through various locations around the world. Scoring is shown after every round when the player is given a certain amount of money for completing the round successfully. The game is able to tabulate the total number of kills, along with where on the enemy's body the bullet hit.

Players can compete in multi-player scenarios similar to Quake III Arena™. These scenarios include CTF and various death match games played on specifically designed maps. The only notion of cooperative teamplay is in the CTF game, where teams compete to return the opposing team's flag to their own base. There are no built-in commands for directing the actions of other humans in the game. You can only use the chat feature to type in your own commands to task other players.

Soldier of Fortune® was written and developed by Raven Software. Official web site: http://www.soldier-of-fortune.com/

12. Spec Ops™: Ranger Gold

In Spec Ops™: Ranger Gold, the user is thrust into a squad of elite soldiers that fight through various missions. The player is able to lead a small squad by issuing

formation commands to the other computer-controlled members. There are no multi-player scenarios in this game, so the teamplay aspects are lacking. Players are scored by the number of kills they get, along with the successful completion of a mission.

Spec Ops™: Ranger Gold was written and developed by ripcord Games. Official web site: http://www.ripcordgames.com/games/specops/index.html

13. Starsiege™ Tribes

Starsiege™ Tribes is mostly a FPS type of game. It offers a third person view, a "buddy" view, and a commander's view, but it is primarily played in first person and third person. The Buddy View presents the player with the first person view of another player, so that he or she may monitor that player. The Commander's view presents the player with a two-dimensional view of the playing area. Starsiege™ Tribes is played like most other FPS games in that the game is most often played as team versus team, or as a death match free-for-all.

The team missions encourage players to coordinate their efforts to accomplish a predetermined goal. The coordination requires effective communication among the team members. Starsiege™ Tribes is the only game that we evaluated that has a leadership hierarchy. Commanders can give orders to subordinate commanders that have command over their subordinates. Commanders have access to the Commander's view, which allows the commander to set waypoints and objectives for subordinates.

Starsiege™ Tribes uses a proprietary game engine - an upgrade of their original Tribes Engine.

Starsiege™ Tribes was written and developed by Dynamix (owned by Sierra). Official web site: http://www.starsiege.com/home.html

14. SWAT™ 3

SWAT™ 3 currently does not support multi-player games, however, there is a multi-player game being developed. In spite of this, SWAT™ 3 does offer some interesting teamwork concepts. In SWAT™ 3, the player plays the role of a team leader. The player directs the other members of the team to meet the objective of the mission. Other than directing the subordinates, there is no other concept of leadership nor is there any concept of a higher level in the chain of command. After completion of the mission, the player reports to a higher command the success or failure of the mission, but no circumstances require the player to coordinate with the higher command during gameplay.

Some aspects of SWAT™ 3 can be modified. It is possible to create new maps and character skins. The user interface can also be changed some. The source code is not available to consumers.

SWAT™ 3 was written and developed by Sierra Studios. Official web site: http://www.sierrastudios.com/games/swat3/

15. Thief™ Gold

Thief™ Gold is written as a single-player game with no networking capabilities. Users may add their own maps to the game, but they cannot affect the play of the game. There is no teamplay and no way for players to cooperate with each other.

Thief™ Gold was written and developed by Eidos Interactive. Official web site: http://www.eidosinteractive.com/games/info.html?gmid=47

16. Unreal™ Tournament

As a first person shooter game, Unreal™ Tournament (*UT*) is not much different than other FPS games such as Quake and Half-Life, especially when playing a multi-player game. UT also offers a few "team versus team" multiplayer games that have not been in other FPS games. One game type called "Domination" has each team fight for possession of a few control points scattered throughout the map. Another game called "Assault" places each player in one of two teams – attackers and defenders. The attackers are given an objective, such as blow up a computer terminal, and it is the job of the defenders to prevent the attackers from completing the mission. Each team is timed to see how long it takes them to complete the mission. The team with the shortest time wins.

Unreal™ Tournament is a very configurable game. Many different options are available for modification. Some of the source code has been released to further enhance the potential for future mods. There have been many excellent mods written so far.

Unreal™ Tournament was written and developed by Epic Games. Official web site: http://www.unrealtournament.com/

17. The Sims™

The Sims™ is a single-player game that allows the user to control the behavior of a family of computer-generated characters. The user assumes a God-like role over the life of each family member. This allows the player to alter the family members' behaviors. There is no networking or teamplay in this game, since the player is alone in controlling the given set of characters. No score or goal is given to the user, so the game never really ends. The fun is in the playing of the game, not the reaching of a goal.

The Sims™ was written and developed by Maxis (a division Electronic Arts, Inc). Official web site: http://www.thesims.com/index.phtml

18. Microsoft® Allegiance™

Microsoft® Allegiance™ is a game a lot like Team Fortress in that players form teams composed of different classes, or player types. Teams have commanders that have the capability to issue orders to team members. Microsoft® Allegiance™ is the only game reviewed here that actually forces the behavior and gameplay to change when orders are received. For instance, if a commander gives and order to a subordinate to navigate to a defense point, once those orders are acknowledged, that player will navigate to the waypoint. Allegiance™ offers a FPS view for combat and two-dimensional map views for navigation. Allegiance™ games can have hundreds of players at one time. There are various goals for each different game.

Official web site: http://www.microsoft.com/games/products.asp?filter=allegiance

19. Sammy Sosa High Heat Baseball

Sammy Sosa High Heat Baseball is the only sports themed game included in this evaluation. Because baseball is a team sport, we want to see how well they had implemented teamplay in the video game. The game can be played one-on-one against another player over a network. Each (human) player plays the role of either pitcher or batter, depending on whether his/her team was batting or fielding. The other members of each team are either computer controlled bots or can be controlled by the human player. If the human chooses to control the other players as well, they can control the player

movements, base running and throwing. There are no other real interactions with other members of the team.

Sammy Sosa High Heat Baseball does not have the capability to be modified by consumers. Consumers can alter characteristics of the game and players to a certain extent, but there are no provisions to alter the rules, create new ballparks, or the like.

Sammy Sosa High Heat Baseball was written and developed by 3DO. Official web site: http://www.3do.com/products/pc/hh2001/index.html

20. iM1A2 Abrams™

Interactive Magic's iM1A2 Abrams™ places the player inside of a US Army Abrams Main Battle Tank. Players can assume one of four positions – Tank Commander (buttoned or unbuttoned), Gunner, and Driver. Players also have the option to control up to six tanks in a combat simulation. Networked games can either be played head-to-head, or cooperatively. There are no provisions for further modification to iM1A2 Abrams™.

iM1A2 Abrams™ was written and developed by Interactive Magic. Official web site: http://www.imagic.com

	Networked Gameplay? (**Y**es or **N**o)	Cooperative Gameplay? (**Y**es or **N**o)	Teammates (**C**omputer, **H**uman, **B**oth, **N**one)	Communication (**T**ext, **V**erbal, **B**oth, **N**one)	Commands (**U**ser, **C**anned, **B**oth, **N**one)	Detail of Commands (**1** [Low] … **5** [High])	Command of Humans? (**Y**es or **N**o)	Command of Bots? (**Y**es or **N**o)	Max # of Players	Concept of Team Leaders/Followers? (**Y**es or **N**o)	User Able to Lead (**C**omputer, **H**uman, **B**oth, **N**one)	Hierarchical Leadership? (**Y**es or **N**o)	Are Players Evaluated/Scored? (**Y**es or **N**o)	Are Teams Evaluated/Scored? (**Y**es or **N**o)	Create Maps? (**Y**es or **N**o)	Source Code Available? (**Y**es or **N**o)	Modify Bot Behaviors? (**Y**es or **N**o)	Modify Gameplay? (**Y**es or **N**o)
Asheron's Call™	Y	Y	H	T	U	1	N	N	1000+	Y	H	Y	Y	N	N	N	N	N
EverQuest™	Y	Y	H	T	U	1	N	N	1000+	Y	H	N	Y	N	N	N	N	N
Ultima™ Online: The Second Age	Y	Y	B	T	B	2	N	Y	1000+	Y	C	N	Y	N	N	N	N	N
Delta Force™ 2	Y	Y	B	B	U	1	N	N	50	N	N	N	Y	Y	N	N	N	N
Half-Life®: Opposing Force™	Y	Y	B	T	U	1	N	N	32	N	N	N	Y	Y	Y	Y	Y	Y
Counter-Strike	Y	Y	H	T	B	3	N	N	32	N	N	N	Y	Y	Y	N	N	N
Team Fortress Classic	Y	Y	H	T	B	3	N	N	32	N	N	N	Y	Y	Y	N	N	N
Quake III Arena™	Y	Y	B	T	B	2	N	Y	32	Y	C	N	Y	Y	Y	Y	Y	Y
Tom Clancy's Rainbow Six	Y	Y	B	T	U	1	N	Y	8	Y	B	N	Y	Y	N	N	N	N
Tom Clancy's Rainbow Six Rogue Spear ™	Y	Y	B	T	U	1	N	Y	8	Y	B	N	Y	Y	N	N	N	N

Table 1: Summary of Game Evaluations

	Soldier of Fortune®	Spec Ops™: Ranger Gold	Starsiege™ Tribes	SWAT™ 3	Thief™ Gold	Unreal™ Tournament	The Sims™	Microsoft® Allegiance™	Sammy Sosa High Heat Baseball	iM1A2 Abrams™
Networked Gameplay? (**Y**es or **N**o)	Y	N	Y	N	N	Y	N	Y	Y	Y
Cooperative Gameplay? (**Y**es or **N**o)	Y	Y	Y	Y	N	Y	N	Y	N	Y
Teammates (**C**omputer, **H**uman, **B**oth, **N**one)	H	C	H	C	NA	B	N	H	N	H
Communication (**T**ext, **V**erbal, **B**oth, **N**one)	T	N	T	T	N	T	N	T	T	T
Commands (**U**ser, **C**anned, **B**oth, **N**one)	U	C	B	C	N	U	N	B	N	U
Detail of Commands (**1** [Low] … **5** [High])	1	2	3	4	NA	1	N/A	4	N/A	1
Command of Humans? (**Y**es or **N**o)	N	N	N	N	N	N	N	Y	N	N
Command of Bots? (**Y**es or **N**o)	N	Y	N	Y	N	N	Y	N	N	N
Max # of Players	32	1	1000+	NA	NA	16	1	100+	2	8
Concept of Team Leaders/Followers? (**Y**es or **N**o)	N	Y	Y	Y	N	N	N	Y	N	Y
User Able to Lead (**C**omputer, **H**uman, **B**oth, **N**one)	N	C	H	C	N	N	N	H	N	H
Hierarchical Leadership? (**Y**es or **N**o)	N	N	Y	N	N	N	N	N	N	N
Are Players Evaluated/Scored? (**Y**es or **N**o)	Y	Y	Y	Y	Y	Y	N	Y	Y	Y
Are Teams Evaluated/Scored? (**Y**es or **N**o)	Y	Y	Y	Y	N	Y	N	Y	N	Y
Create Maps? (**Y**es or **N**o)	Y	N	N	N	Y	Y	N	N	N	N
Source Code Available? (**Y**es or **N**o)	Y	N	N	N	N	Y	N	N	N	N
Modify Bot Behaviors? (**Y**es or **N**o)	Y	N	N	N	N	Y	N	N	N	N
Modify Gameplay? (**Y**es or **N**o)	Y	N	N	N	N	Y	N	N	N	N

Table 1: cont.

D. EVALUATION OF GAME ENGINES

As described in Chapter II, the game engine is the part of the game that displays the player's view onto the screen. When developing a game today, authors have a few choices regarding the game engine. They can write and create their own engine, or they can buy a game engine from another party. Because game engines are very complex, they can take a very long time to develop. For this reason, it is becoming very popular to purchase the game engine from another source.

Popular game engines today are Quake II, Quake III, Unreal Tournament, Half-Life, Renderware, the LithTech™ engine by Monolith, and the NetImmerse Game Engine by Numerical Designs Limited (*NDL*). The NetImmerse engine is the only one of these that was not developed specifically for a particular game. They all offer good performance and are flexible enough to be used in a wide variety of applications. The Quake III and UT engines are relatively new and as such, no games have been released yet using those engines, but there are several games in development.

The Quake II engine was the most used among the games evaluated here. It is a very capable, high-performance engine that can be easily modified to meet many needs. Half-Life® is possibly the most famous of all games that used the Quake II engine. Sierra Software heavily modified the engine to meet their needs. Half-Life® won many awards for sales and innovation.

The Quake III engine is id Software's newest engine. It is currently being used to develop several new games. It offers much greater performance than the Quake II engine, and has new features that have not been present in any previous engine, such as the ability to render parametric curves.

The UT engine is also a new entry into the game engine market. It is currently being used to develop several new games. The UT engine will support Sony's Next Generation PlayStation™. It offers similar performance to the Quake III engine and costs about $500,000 for a royalty-free license and includes technical support for up to twelve months after product release.

The Half-Life® is a heavily modified Quake II game engine. Valve Software changed over seventy percent of the code in the Quake II engine to suit their needs. Some of the largest changes were made to the networking code by optimizing and increasing security greatly. The engine is available for licensing and purchase from Valve Software, however, no outside organizations have done so yet. Valve has licensed the game engine to other companies to create Half-Life® mods for Valve, and a few non-gaming projects. No pricing information is available for the engine.

An interesting choice among the field of game engines is the RenderWare 3 game engine by Criterion Software, Ltd (http://www.renderware.com). It is a relatively new engine (about one year old) available at a very low cost (starting at only $1000). Technical support options are available at additional cost. No games have been released using the RenderWare 3 engine, so it could not be evaluated on its performance.

The LithTech™ Engine by Monolith (http://www.lith.com) is also a relatively new engine. Licenses start at about $250,000. Two games have been released using this engine (Shogo and Blood 2) and the source code to these games is available to developers that purchase a license. The engine comes with level and model editors to make the creation of the three-dimensional world easier. The engine also includes networking features (the RenderWare 3 and NetImmerse engines do not include networking).

Numerical Design Limited (http://www.ndl.com) developed the NetImmerse game engine specifically for resale to other parties. Their engine is more flexible than the engines that were developed for specific games, such as Quake, because their goal is to make the engine suitable for many different customers. The engine is inexpensive (about $75,000) compared to other engines available today, and includes technical support. Several popular games have been developed using the NetImmerse engine, and many more are in development. The NetImmerse engine is also capable of producing games for other platforms, such as game consoles and Apple computers.

E. FINAL CANDIDATES FOR RE-PURPOSING

None of the games evaluated here met all of the requirements that we needed. Most of the games did not lend themselves to re-purposing due to the fact that they were not modifiable by the consumers. The few FPS games that were modifiable included Quake III Arena™, Half-Life®, and Unreal Tournament™. Because of this fact, we limited our final candidates to these three games. Chapter IV includes a detailed analysis of our final decision process.

IV. PROOF OF CONCEPT

This chapter outlines the steps required to build a modification of a commercial game. Someone creating mods for other games should expect to take steps similar to those outlined in this chapter. Also listed in this chapter is the basic design of our mod to Quake III Arena™.

A. INTRODUCTION

Our goal was to determine how much difficulty is involved in creating a mod that could theoretically be used by the military for some useful purpose. The results of this study could then be used to predict whether or not a re-purposing effort of existing software is feasible for other similar projects. Along these same lines, we also wanted to determine the expected cost, and how much time would take to develop that mod. We chose to develop a proof-of-concept application by re-utilizing one of the games evaluated in Chapter III. We wanted to create a useful application by modifying an existing game using available tools, or by creating simple tools if needed, to create a new application that provides a useful purpose.

The scenario of "Assault on Herrmann Hall" places users on a counter-terrorist team with the objective of rescuing a hostage from terrorists in a real building. The objective will be difficult enough that it will encourage teamwork by requiring players to formulate a plan and execute that plan.

1. Mod Development Tools

In order to create a modification of a game, special tools (software applications) are required. Common tools used in creating mods today are level editors, modeling

tools, and development environments. Level editors are used to create the maps, or worlds, or environments that the game takes place in. They typically have a three-dimensional editing and viewing capability. These tools also have texturing capabilities for the level's objects. Modeling tools are used to create many of the other entities within the game, such as weapons, other objects, and characters. One popular such tool is 3D Studio Max®, but other tools are available. Finally, development environments are used to modify source code. Actually, a development environment is not required, *per se*, but they do make the development process much easier. Some people prefer to edit source code in a stand-alone text editor and compile the source from outside the editor. A development environment simplifies the process by integrating the editor with the compiler.

2. Why We Chose Quake III Arena™

Based on the game evaluation criteria, we eliminated many of the games evaluated in Chapter III from consideration. The games we determined to be the best candidates for a re-utilization effort were Quake III Arena™, Half-Life®, Unreal™ Tournament, and Soldier of Fortune®. We knew that we wanted to incorporate a model of Herrmann Hall (a large, architecturally interesting building on the NPS campus) into whichever game that was selected. All games have limitations as to how large their maps can be; of the above games, Q3A is able to render the largest maps. The source code would also have to be available for the game chosen because of the level of modifications that would be needed. Q3A and Half-Life® both make their source code publicly available. Because the Q3A engine can render a much larger map than Half-Life's, we decided to use Q3A for our re-purposing effort proof-of-concept.

Q3A's game engine is also capable of rendering parametric curves and surfaces. While this was not a requirement for this thesis, Herrmann Hall does have some arches that add significantly to visual presentation.

B. STEPS IN MOD CREATION

The following is a step-by-step tutorial that is designed to show how to create a game mod. The basics are explained here, using our mock application as a running example; the details that are specific to our implementation are explained in Chapter V. We have listed the time in man-hours that it took us to complete each step in the mod process. These times will vary based on the complexity of the mod that is being built.

1. Scenario Design

The first step in mod creation is to determine the basic scenario. We chose a hostage scenario, but one could envision many different types of games that could be designed.

Our mod of Q3A, called "Assault on Herrmann Hall", is relatively straightforward. A hostage has been taken and is being held captive somewhere in Herrmann Hall. It is up to the human players to rescue the hostage. The scenario included multiple computer-controlled opponents to hunt down the human players, making for a challenging game.

This phase took us approximately ten man-hours to complete.

2. Level Design

Once the design of the scenario was set, it was time to design a level to play it in. Level Design is vital to the success of any modification. There are two basic types of levels that can be used: those that are imported and those that are created from scratch.

Levels created from scratch can be made into anything the designer desires. These levels are usually fantasy-based with no attempt at realism. A professional level designer, with an approved design, can generally create a finished level in three or four weeks. The design of a pleasing and technically sound level could take weeks or months by itself.

Imported levels are usually real buildings that are converted into the appropriate format for the game that is being modified. For these levels, realism is important. We chose to import a model of an existing building because realism was key for our scenario. Because it was the intention that this application serve as a team-trainer, it was necessary that the scenario setting be as realistic as possible. A hostage scenario in a gothic fantasy world simply wouldn't be believable.

We had an AutoCAD® model of Herrmann Hall, but it was not compatible with the Quake map format. There were no programs or converters to assist placing real building models into Q3A. We then proceeded to write a converter program to convert the AutoCAD® model into the Quake map format. Since the AutoCAD® model did not contain texture information, the model would have to be manually textured. By showing that Q3A can indeed import an AutoCAD® model, we have demonstrated an alternative to the slow and expensive current options.

Herrmann Hall is a very large building, and the computer-controlled kidnappers are very skilled. A single player acting alone would not very likely be able to rescue the hostage. Therefore, the players are encouraged to coordinate with other players to increase their chance of success. Because Herrmann Hall is so large, we limited the gameplay to the first three floors of the building. This keeps the gameplay faster by preventing the player from searching the entire building for hostages and kidnappers. It also congregates the kidnappers more, allowing them to work together more. One issue this brings to light is the interaction between each of the steps involved in creating a mod. Limiting the gameplay to the first three floors is both a level design issue and a gameplay issue. Working in one design phase almost always leads to issues in another design phase.

The model conversion took a total of 100 man-hours. This included 10 man-hours to write the converter. The time to import a new building would be lower since the converter is already written and we learned a lot to help speed up the process.

3. Modeling

Once the level is complete for the scenario, changes have to be made to the in-game models. Some mods require new characters to be added, along with their skins and animations. Other mods need to add new weapons or other useful models. What these models have in common is their requirement to be built using a special tool such as 3D Studio Max®.

We chose to make specific changes to Q3A that would help make our scenario more realistic. First, we added a laser designator to the weapons arsenal that each player carries. That model required a texture, so we found a texture online that we could use.

The hostage needed a special skin color, so we located his texture files and changed his skin color as necessary. We didn't want the hostage to be able to carry any weapons, so we rendered his only weapon invisible. Now that the hostage was unable to fire any weapons, we needed to change his behavior from raging killer to passive hostage.

The time to build the models and put them into Q3A was approximately 10 man-hours. Again, the time to build new models decreases as each one is built. An expert model builder could probably build them much faster than we did, since we started with no model building experience.

4. Artificial Intelligence

Once the models are completed, it is often the case that the mod requires a change in the behavior of some of the bots. If new bots are added, then their behaviors must be created from scratch. Q3A controls bot behavior using specific C++ files that designate values for certain parameters that control a bot's characteristics.

We needed the hostage to act benign, so its AI had to be reprogrammed to keep him from attacking other players and chasing after weapons. The hostage's behavior was changed so that he had no aggressiveness, but had a great desire for self-preservation.

Our scenario requires that the other bots roam the world looking for our human players to kill. When the mod was first created, the bots would only stay in one spot instead of wandering freely. We had to add weapons around the world in order to provide the bots with some impetus for moving. This was a problem for realism though, because now we had a lot of unrealistic weapons lying around. We were forced to render these weapons invisible, so that the players cannot see the weapons that the bots are searching for.

The time to change Biker's AI code and insert it into Q3A was 10 man-hours.

5. Gameplay

After the models have been added and the Artificial Intelligence is set, the next step is to make changes to the gameplay. Some mods keep the same gameplay as the original game, and will not require any effort at this stage. Others will require major changes, since the mod is not the same as the original game. We needed to make very few changes to the gameplay, since our scenario was closely related to the original game.

The biggest change to the gameplay made for this mod is giving each player only one life. In Q3A, each player is reincarnated in the world after being killed. Because we wanted our game to be more realistic, each player and bot is given only one life for each game. To accomplish this change, the source code had to be modified. Decisions had to be made to determine exactly what does happen to a player when they are killed. This could range from ending the game immediately, or perhaps better, presenting the player with feedback about how they did and offer them a chance to play again.

Another gameplay change that needed to be made was the removal of some of the sounds in the game. For example, the laser designator replaced the railgun in the player's inventory. This meant that whenever the laser designator was "fired", the sound of the railgun shooting played. To remove this sound, we used a file included in Q3A called *silence.wav*. The *silence.wav* file replaced the original railgun sound file so that no sound was actually played.

The time required to change the gameplay was 15 man-hours. More complex changes in gameplay would require more time to implement.

6. User Interface

The last step in creating a mod is to change the User Interface (UI) to incorporate necessary changes to the game. This can include changing menus, screen appearance, player text messages, etc.

Our mod, if developed into a team-trainer, would require the functionality to monitor and record various aspects of how well each member contributes to the team. We chose to implement this by presenting the player with a menu that allows them to record a rating for other players in the game. To accomplish this change, modifications to the user interface sections of the source code were necessary.

The time required to alter the User Interface was 20 man-hours.

V. IMPLEMENTATION

This chapter details the entire modification process for our mod of Quake III Arena, called "Assault on Herrmann Hall". These step were summarized in Chapter IV. This chapter explains the tools used, where to get the tools, and the steps required to complete each facet of the mod.

A. HERRMANN HALL MODEL

"Assault on Herrmann Hall" required us to develop a method to incorporate a realistic looking building as the game level. This could be done two ways—either modeling the entire building from scratch, or by importing an AutoCAD® model of Herrmann Hall into the Quake map format. To accomplish this step, we wrote a converter to speed up the development.

1. Converter

AutoCAD® uses a text file format, which can easily be read by a C++ program. The Quake map format is also a text file format. AutoCAD® stores its databases in what is known as a "dxf" file. The dxf file stores vertex information for each three-dimensional plane in the database. The Quake map format stores "planar" information for each three-dimensional plane in the model. In other words, instead of storing each vertex of the plane to define its boundaries and edges, three points that lie on the plane are stored. It is a geometric law that any three non-collinear points will define a plane in three-dimensional space. Three vertices may be used to define the plane, but the points do not have to be vertices. In fact, the points do not even have to lie on the plane, as bounded by the other edges of the three-dimensional solid. Figure 2 illustrates the use of

vertices and planar points to define a plane in each format. The planes edges are defined by the intersection of the other planes on the three-dimensional solid. Additionally, all "pseudo-planes" in Quake are actually three-dimensional solid objects. Therefore, a model of a six-sided room would actually be composed of six separate solid objects. Figure 3 illustrates the difference between Quake's solids and AutoCAD's® polygons.

The converter accomplishes several tasks. First it reads the dxf file into memory. Next, it parses out the vertex information for each plane and converts that information into 3-point planar information. Next, it converts that plane into a solid by extruding the plane a minimal distance (in Q3A, one unit distance) along the plane's surface normal and recording the planar information about each of the five newly generated planes. Figure 4 illustrates this step. Once the planar points have been determined, it is next necessary to place them in clockwise order, as viewed from outside the solid. Finally, the information for each solid is saved to a file in the correct syntactical format (for example, parenthesis and carriage returns placed correctly).

The converter code written accomplishes all of these tasks. It was written in C++ and is relatively straightforward. The full source code is available in Appendix A.

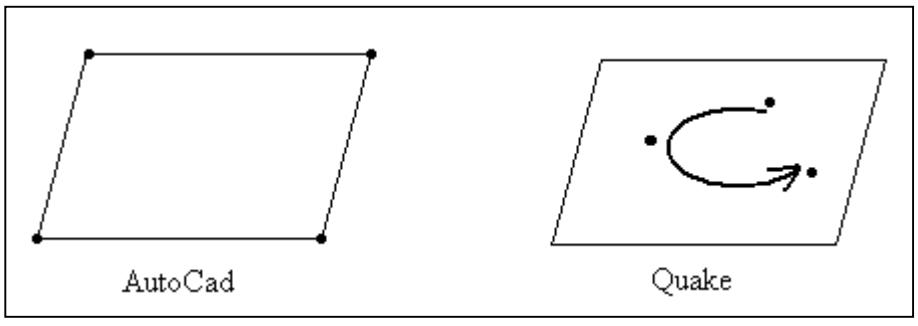

Figure 2. AutoCAD® versus Quake® Plane Definitions

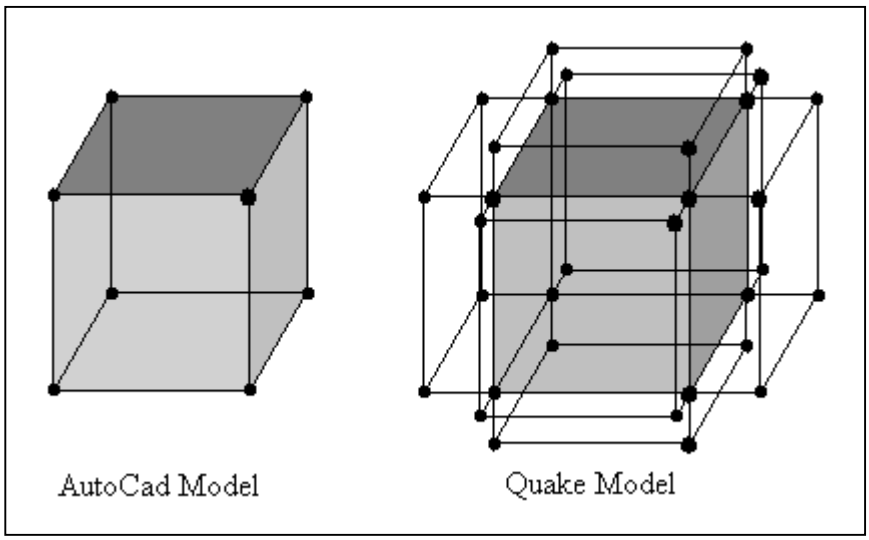

Figure 3. AutoCAD® versus Quake® Geometry

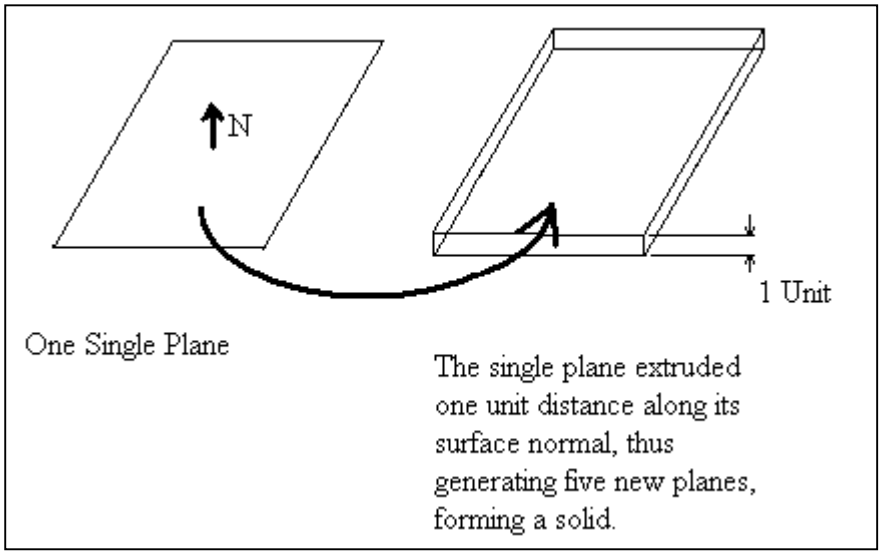

Figure 4. Extruding a Plane to Form a Solid

The converter took about one week to code and was very effective in converting the Herrmann Hall AutoCAD® model into the Quake map format. We feel that attempting to model Herrmann Hall by hand without this conversion program would have been impossible. The converter has been tested with some other small AutoCAD® models, and was also effective in converting them. The conversion program takes less than one minute to execute and convert the Herrmann Hall model, which we consider to

be close to the absolute maximum size that the game engine could handle. Other conversions should take similar times to complete. The conversion process produced a few artifacts with the Herrmann Hall model. These artifacts were not considered severe, and were easily remedied during the texturing process. These artifacts are discussed in greater detail in the next section, "QERadiant".

2. QERadiant (Optimization and Texturing)

Once the model conversion was completed, the next step was texturing and model optimization. We had access to digital photographs of various parts of Herrmann Hall, which could be used as textures. Of all the steps necessary to complete the model, texturing took the longest. Figure 5 shows a typical screenshot of QERadiant, the modeling and texturing program used to create the Herrmann Hall model.

As discussed in the previous section, the conversion process created a few artifacts, usually in the form of additional polygons or intersecting polygons. The problem with extra polygons and intersecting polygons is generally either a performance decrease or a visual error. For example, when polygons intersect, only one polygon will be rendered at any given point in space, even if there is more than one polygon occupying that space. Therefore, the game engine has to make a decision as to which polygon to render last–that is, decide which polygon is actually seen by the player. Sometimes this is seen as a shimmering effect, which is visually distracting and decreases performance. When intersecting polygons are eliminated, the map plays more efficiently. Similarly, sometimes polygons are created in spaces too small for a player to fit into. The game engine still completes all visibility, lighting, and collision detection for these polygons as well, again wasting resources.

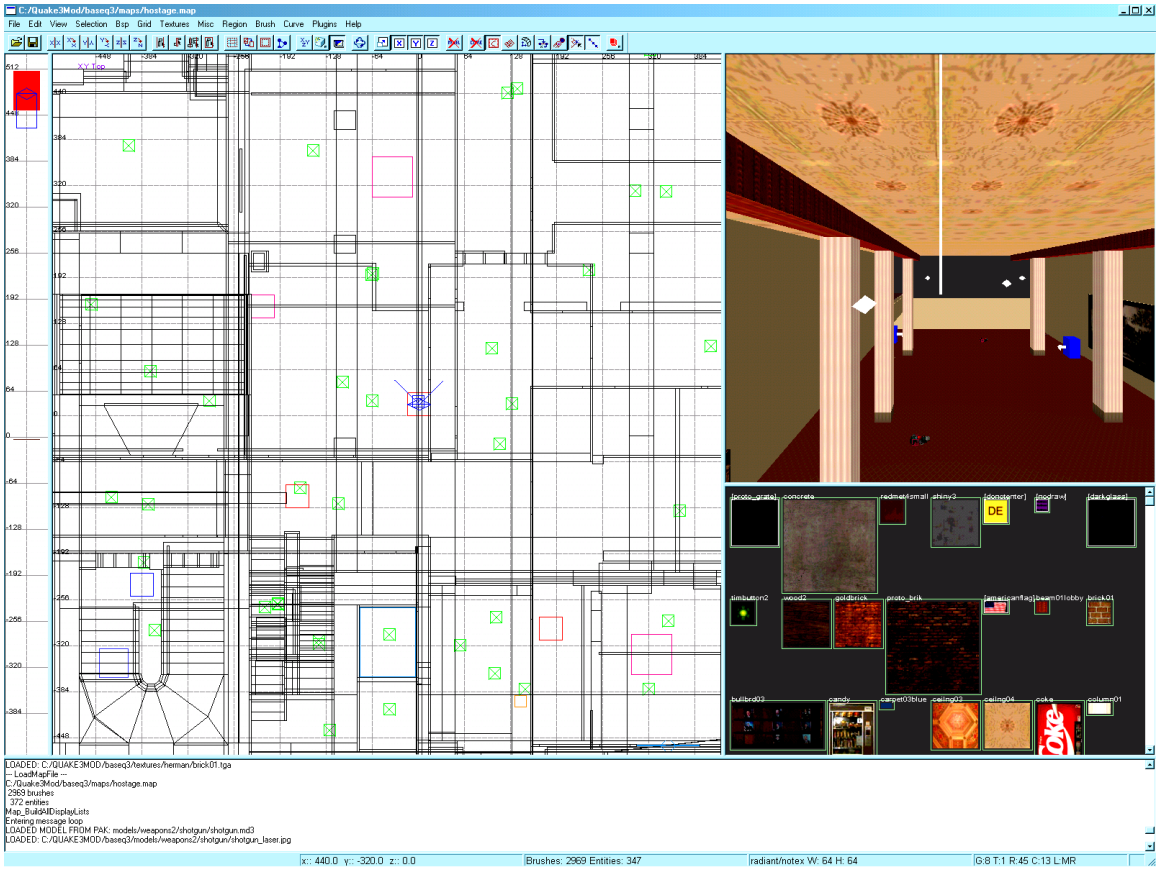

Figure 5. QERadiant

One other type of artifact created by the conversion process is non-complete enclosure within rooms and spaces. If two polygons are placed close to each other, but not quite exactly against each other, the engine will attempt to "look" into the space created by this gap. Even though the gap may be too small for a player to fit into, (or even look into,) that small gap is still rendered by the engine, which ultimately decreases performance.

It is unknown what the exact cause of the artifacts is. They could either be a result of inaccuracies in the AutoCAD® model, or by the conversion process itself. Because it was very easy to find the artifacts and correct them, this was not considered to be severely detrimental to the overall process. Certainly, the small amount of time spent

correcting the artifacts greatly outweighs the amount of time it would have taken to model Herrmann Hall from scratch, and not have any artifacts.

One other important consideration is the limitation of the game engine on maximum map size and its capability to render large "outdoor" scenes. The Herrmann Hall model is very large (approximately 200 meters by 200 meters in real life). Quake maps limit all coordinates in x-y-z space to +/- 4096 units. The Herrmann Hall map as imported would not fit within those coordinates. Scaling was not an option since hallways and doors must accommodate the players and bots. Therefore, we were forced to remove the two administration wings from the back of the model. The original AutoCAD® models did not have any interior geometry in the administration wings to begin with, so removing them was not seriously detrimental to realism.

Q3A's engine uses binary space partitions (BSP) to speed up rendering and keep the frame rate at an acceptable level. The way a BSP engine works is to first compute a potentially visible set (PVS) by removing objects that cannot possibly be seen from the current viewpoint. In complex indoor scenes, where the only objects in the PVS are those that are located in the room, and maybe a few objects located right outside the door, the majority of the map will not be passed down the rendering pipeline. However, large, outdoor scenes force the BSP calculations to include almost the entire map, eliminating only objects inside the building. Due to this difficulty in rendering large outdoor scenes, we chose to eliminate all outdoor scenes from the final "Assault on Herrmann Hall" mod.

The final step after creating and texturing the model in QERadiant is compiling the map. This involves running a compiler on the *.map file just created. The compiler accomplishes two important functions for the game engine. First, the compiler converts

the map into binary space partitions. Second, the static lighting for the model is all pre-computed. By pre-computing as much of this information as possible, the map can be drawn in real time with an acceptable frame rate. The compiling process can take from less than a minute for a well-designed map to several days for maps with large outdoor areas, such as the Herrmann Hall model.

B. QUAKE PAK FILE FORMAT

Q3A allows users to change source code in their game and then use that code to play a modified game. The source code for bot intelligence, game models, skins, and textures is contained in a file called *pak0.pk3*. The file is compressed using the PKZIP algorithm and, when expanded, contains over 60 megabytes of data in 3,357 files. This makes the file rather hard to work with, especially when making multiple changes. Also, it is considered dangerous to actually modify your *pak0.pk3* file, because this file must be identical to every other player's *pak0.pk3* file in order to have networked game play.

The way that Quake has conquered this problem is to allow users to create their own pk3 files. When the game starts, Q3A searches for all pk3 files that are in the root game directory. The engine starts with pak0 and counts up sequentially from there (pak1.pk3, pak2.pk3, etc.). The benefit is that files that are in a later pk3 will replace files that are in an earlier pk3, if they have identical names. For example, any bot code that we changed went into a file called *pak3.pk3*. While these files are also contained in the unmodified *pak0.pk3* file, they are essentially replaced when our new file is read. This method of inserting modified files makes changing the game much easier, along with identifying changes that have been made.

C. ARTIFICIAL INTELLIGENCE

In order to have a believable hostage scenario, we had to have a hostage in the game. This hostage had to be totally passive, since a real hostage would not be likely to attack his captors with guns blazing. Also, the hostage had to stay in one place. None of the bots that were included in the game fit this profile, so we had to create our own hostage by changing the Artificial Intelligence (AI) of one of the bots. No other bot's AI was changed, since the team designation in CTF allows the bots to act as enemies.

We chose to modify the behavior of Biker, a particularly large and disgusting bot in Q3A. He was easy to tell apart from other bots because of his size. Unlike many Q3A bots, Biker looks human, which helped to keep the mod somewhat realistic.

The source code in Q3A contains four files for each bot to control their behavior. These files are written in C. The entire original and modified files are available in Appendix B. For the purposes of this explanation, the filenames for Biker will be used:

- *Biker_c.c*: gives all of the values to Biker's characteristics
- *Biker_i.c*: gives the weapon and power-up weights for Biker's inventory
- *Biker_t.c*: contains all the chat lines for Biker to use during the game
- *Biker_w.c*: contains the other weights for the weapons used by Biker during the game

First we took the file *Biker_c.c* and figured out the purpose of all of the characteristics. See Figure 6 for a subset of the characteristics that are contained in this file. All of the numbers in Figures 6 and 7 are relative numbers between 0.0 and 1.0.

```
CHARACTERISTIC_CROUCHER             0.5
CHARACTERISTIC_JUMPER               0.5
CHARACTERISTIC_WEAPONJUMPING        0.5
CHARACTERISTIC_GRAPPLE_USER         0.5
CHARACTERISTIC_ITEMWEIGHTS          "bots/biker_i.c"
CHARACTERISTIC_AGGRESSION           0.5
CHARACTERISTIC_SELFPRESERVATION     0.5
CHARACTERISTIC_VENGEFULNESS         0.5
CHARACTERISTIC_CAMPER               0.5
CHARACTERISTIC_EASY_FRAGGER         0.5
CHARACTERISTIC_ALERTNESS            0.5
```

Figure 6. Example of Biker's Unmodified Characteristics

```
CHARACTERISTIC_CROUCHER             0.0
CHARACTERISTIC_JUMPER               0.0
CHARACTERISTIC_WEAPONJUMPING        0.0
CHARACTERISTIC_GRAPPLE_USER         0.0
CHARACTERISTIC_ITEMWEIGHTS          "bots/biker_i.c"
CHARACTERISTIC_AGGRESSION           0.0
CHARACTERISTIC_SELFPRESERVATION     1.0
CHARACTERISTIC_VENGEFULNESS         0.0
CHARACTERISTIC_CAMPER               0.0
CHARACTERISTIC_EASY_FRAGGER         0.0
CHARACTERISTIC_ALERTNESS            0.0
```

Figure 7. Example of Biker's Modified Characteristics

The unmodified value for CHARACTERISTIC_JUMPER in Figure 6 is 0.5. If this value were 0.0, then Biker would rarely jump while running through the game world. If this value were 1.0, he would jump almost constantly. As the bot's skill level increases, his propensity for jumping tends to increase, since a jumping target is harder to hit. In order to make Biker to appear more like a hostage, we made his jumping value 0.0 (see Figure 7). Another example is the CHARACTERISTIC_AGGRESSION value, which was changed to 0.0 to make Biker less inclined to seek out enemies and attack them. In higher skill levels, bots tend to have a very high aggression level, which makes

them more dangerous to the players. Another important personality characteristic is the CHARACTERISTIC_SELFPRESERVATION. We changed this value to 1.0, which makes Biker very careful about ever attacking anyone. If he is caught in the crossfire of a gunfight, he will run away to help save himself. These simple modifications yield a reasonably accurate reflection of the way a hostage would react in the real world.

Next we looked at the *Biker_i.c* file to determine the purpose of the weapon weights as written in the code. Figure 8 shows the original code with the original weapon values given to Biker. Figure 9 contains the modified code with altered weapon weights.

```
//initial weapon weights
#define W_SHOTGUN                   50
#define W_MACHINEGUN                40
#define W_GRENADELAUNCHER           40
#define W_ROCKETLAUNCHER            120
#define W_RAILGUN                   85
#define W_BFG10K                    30
#define W_LIGHTNING                 50
#define W_PLASMAGUN                 500
```

Figure 8. Biker's Original Weapon Values

```
//initial weapon weights
#define W_SHOTGUN                   1
#define W_MACHINEGUN                1
#define W_GRENADELAUNCHER           1
#define W_ROCKETLAUNCHER            1
#define W_RAILGUN                   1
#define W_BFG10K                    1
#define W_LIGHTNING                 1
#define W_PLASMAGUN                 1
```

Figure 9. Biker's Modified Weapon Values

The weights have a minimum value of 1 and a maximum of 950. Bots use these weapon weights to determine which weapon to find and subsequently pick up (McDonald, 1999). A higher number for a weapon means the bot prefers to find that

weapon rather than one with a lower number. A large difference between values means that the bot will go out of his way to find the weapon with the higher value, even if it means walking by other available weapons. By giving different bots a preference for different weapons, Q3A ensured that all of the bots in the world would not be looking for and using the same weapons. We changed the values of Biker's weapon weights to all ones, meaning that he would not seek out any specific weapons to use.

This file also contains a second set of weapon weights that are prefaced with the comment, "The bot has the weapons, so the weights change a little bit." These weights are similar to the first set, except each weapon's weight is decreased slightly. The purpose behind a second set of weights is that the bot now has the weapon, so he doesn't need to look for it as much as before. Picking up another will only provide him with ammo for that weapon, so the weight does not need to be as high.

The file *Biker_t.c* contains the lines of text that Biker can "say" during the game. While we didn't need to change anything that Biker said, we wanted to see if it could be done. Figure 10 shows the original lines that Biker would use to taunt the other players during the game. In Figure 11, we added the line "I am a great hostage." We also commented out the other lines that he would use, making certain that our new line was the only valid one. Q3A gives each bot a certain number of lines to choose from in different situations. Our line was put in the greeting section that a bot uses when joining a game. In our modified version, Biker uses his new line when he joins the game.

```
type "game_enter" //initiated when the bot enters the game
      {
            HELLO5;
            "Make this a good day, cuz it'll be your last.";
            "Everyone out of the pool.";
            "You just keep your hands off my hawg, ", 1, ".
Understand?";
            1, ", I'm gonna kick your sorry butt into the next
county.";
            "I squash cockroaches bigger and scarier than you, ",
1, ".";
            "This better not make me late for lunch.";
            // 0 = bot name
      } //end type
```

Figure 10. Biker's Unmodified Chat Lines

```
type "game_enter" //initiated when the bot enters the game
      {
            HELLO5;
            "I am a great hostage.";
/*          "Make this a good day, cuz it'll be your last.";
            "Everyone out of the pool.";
            "You just keep your hands off my hawg, ", 1, ".
Understand?";
            1, ", I'm gonna kick your sorry butt into the next
county.";
            "I squash cockroaches bigger and scarier than you, ",
1, ".";
            "This better not make me late for lunch.";  */
            // 0 = bot name
      } //end type
```

Figure 11. Biker's Modified Chat Lines

Some of the lines have numbers that are not contained inside the quotation marks. These numbers represent other players' names that can be inserted into the line to make the chat seem more realistic. In our proof-of-concept, we turned off the chat to enhance realism, since players should only be able to communicate with each other via radio.

The last file, *Biker_w.c*, contains the weapon weights for Biker that allow him to choose which weapon to use at any given moment. Q3A uses these weights to control

the weapons a bot will use during the game. For example, Biker originally had a relatively large value for the plasma gun (275). This means that if he had it in his inventory, he would use it instead of any other weapon. It is important to note that these weights are not used in isolation. When choosing a weapon to use, a bot will take more than the weapon weight into consideration. A bot's sense of self-preservation may make him choose a less powerful weapon to avoid infliction any damage on himself.

Similar to the previously mentioned weapon values, we changed all the weights to 1 in order to give Biker no weapon preference at all. See Figure 12 and Figure 13 for the unmodified and modified values, respectively.

```
#define W_GAUNTLET                  10
#define W_SHOTGUN                   30
#define W_MACHINEGUN                20
#define W_GRENADELAUNCHER           40
#define W_ROCKETLAUNCHER            100
#define W_RAILGUN                   70
#define W_BFG10K                    95
#define W_LIGHTNING                 80
#define W_PLASMAGUN                 275
#define W_GRAPPLE                   15
```

Figure 12. Biker's Unmodified Alternate Weapon Weights

```
#define W_GAUNTLET                  1
#define W_SHOTGUN                   1
#define W_MACHINEGUN                1
#define W_GRENADELAUNCHER           1
#define W_ROCKETLAUNCHER            1
#define W_RAILGUN                   1
#define W_BFG10K                    1
#define W_LIGHTNING                 1
#define W_PLASMAGUN                 1
#define W_GRAPPLE                   1
```

Figure 13. Biker's Modified Alternate Weapon Weights

D. IN-GAME MODELS

The changes that we made to the models in the game ranged from simple to complex. The most complex was the laser designator, since it closely resembles a flashlight. The weapons that were not to be used by the players or the bots were made invisible, which required some model changes. We took Biker's skin and changed the dominant color from red to yellow, and we also changed Biker's skin to resemble normal clothing as an experiment. All of these changes are explained in greater detail in the following sections.

1. Laser Designator

We wanted to create a new model to be used by the players that would actually have some functionality in the scenario. Since intra-team communication is paramount, we thought to add a laser designator that could be used to show other team members where to go. The game engine takes care of the visibility issues that come from one player pointing a light at a wall and other players being able to see it. We chose to change the model of the railgun, since that weapon shoots a laser-like beam in the real game.

The first step in changing the railgun to a laser designator was to create a new 3-dimensional model to put in the game. This model of the laser designator was built in MilkShape 3D version 1.3.2. This program is available for free for a 30-day trial period, which was ample time to build the model. To build the laser designator, we took a basic sphere and extruded the barrel, then flared the barrel at the end to make the light. Figure 14 shows a screen shot of the final version of the laser designator.

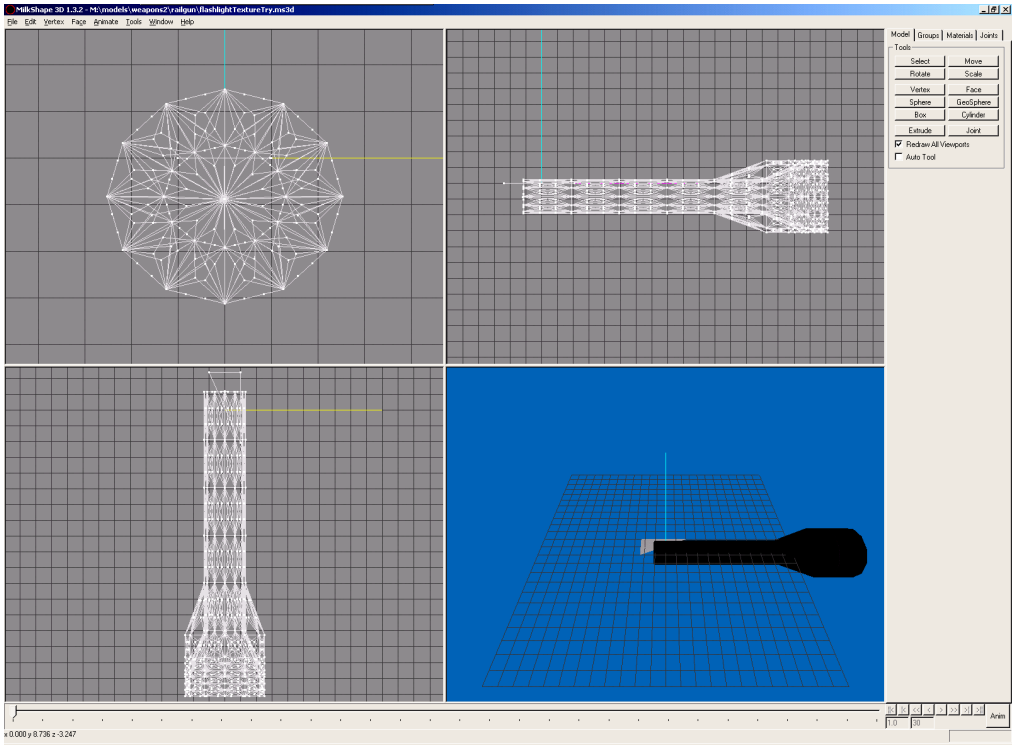

Figure 14. Laser Designator in MilkShape 3D

The unshaded polygons of the model were not very appealing in the game, so we found a texture to apply. Using MilkShape 3D, we were able to apply a texture to the laser designator and make it look much more realistic. The problem then became how to get Q3A to recognize this new texture. The answer was found in MilkShape 3D, through the use of a Quake 3 control file. In order to export a model to Q3A from MilkShape 3D, it is necessary to generate a control file. The name of the control file must exactly match the name of the desired export file in order for the process to work. Figure 15 shows the control file that was generated for the laser designator. Note the texture name, *bluemetalsupport2e256.bmp*, at the bottom of the file. This texture name is then used in the final export file. That way, Q3A knows which texture to apply to the model that is in the file.

```
      // Quake III Arena MD3 control file, generated by
MilkShape 3D
      $model "models/players/model/model.md3"
      // 0 0 = all frames, -1 -1 = reference model, 1 30
= frame 1-30
      $frames 0 0
      $flags 0
      $numskins 0

      // you can have one or no parent tag
      //$parenttag "tag_weapon"

      // tags
      $tag "tag_weapon"

      // meshes (surfaces)
      $mesh "w_railgun1"
      $skin
      "models/weapons2/railgun/bluemetalsupport2e256.bmp
```

Figure 15. Control File for Laser Designator

Also of interest is the line about tags, where it states that the tag being used is called "tag_weapon". Tags are used by Q3A to tell the program where to place the current weapon on the avatar. In order to make the bots and human characters look like they are really carrying the weapons, they all have to fit precisely onto the right hand of the avatar. The weapons have varying sizes, so they need different holding points, and that's where the tag comes in. The tag is placed on the weapon during the modeling phase. In our case, we placed the tag on the laser designator using MilkShape 3D. The tag is just a right triangle that is separate from the model. MilkShape 3D and other modeling programs allow multiple groups to exist in one file. We made a group of all the polygons that made up the laser designator and called it "w_railgun1". This was the same group that Q3A used for the railgun. Since the laser designator was replacing the railgun in our scenario, this naming convention would allow the program to use our model as if it was the railgun. The second group we made was labeled "tag_weapon".

This allowed us to precisely place the designator in the avatar's hand. The first time we tried it, the tag was incorrectly placed so the laser designator ended up above the player's head, rather than in his hand. After some trial and error, we were able to place the tag in the right spot so that the laser designator looks like it was made for the game. Figure 16 shows a picture of the final product in the game.

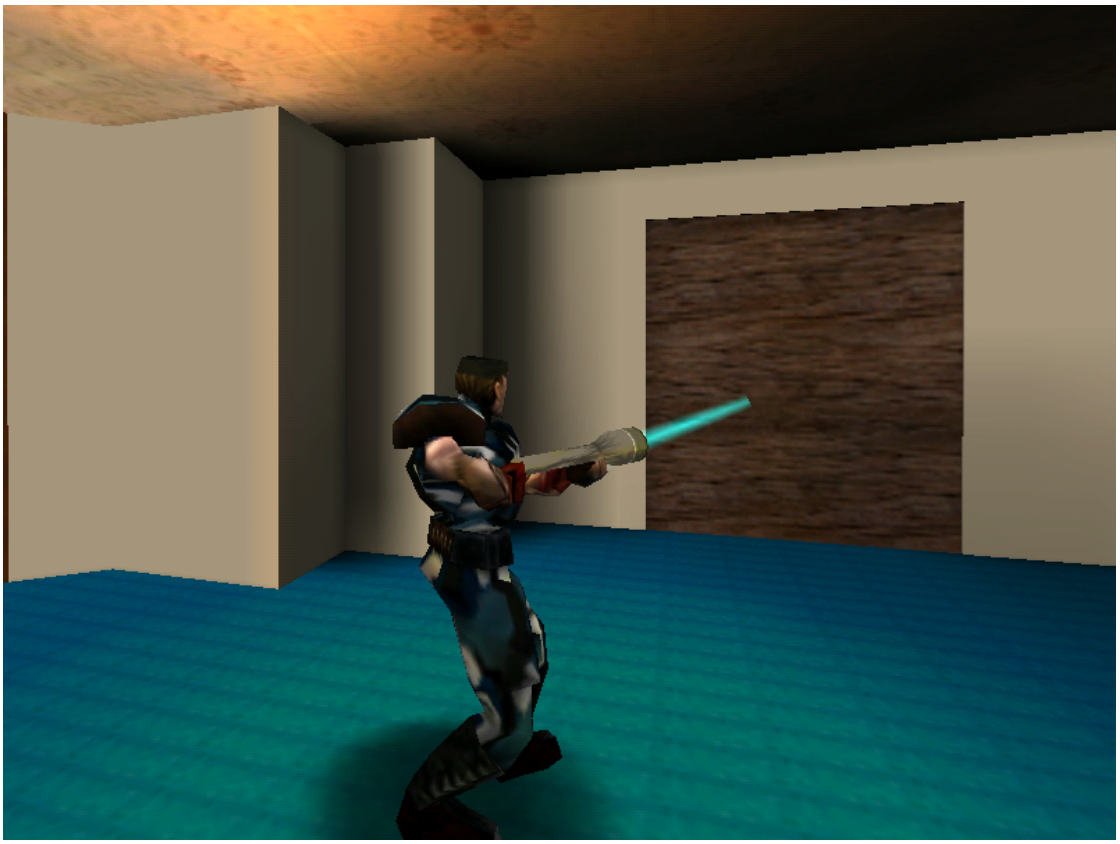

Figure 16. Laser Designator Model in the Game

More information on MilkShape 3D can be found at their official website: http://www.swissquake.ch/chumbalum-soft/ms3d1x/

2. Weapons

Another important step in our modification was removing all weapons that were too unrealistic for a military hostage scenario. While plasma guns and lighting guns

make for a great fantasy game, they are not currently in the Army's inventory, so we had to remove them. This is where our previous discovery of the model tags was helpful. The tag that is used to place a weapon on the avatar is not rendered by the game engine. It is invisible to the players in the game, since it is just used as a positional marker. This could be used to our advantage in trying to make the unwanted weapons disappear. Figure 17 shows the file we created in MilkShape 3D that contained only a tag.

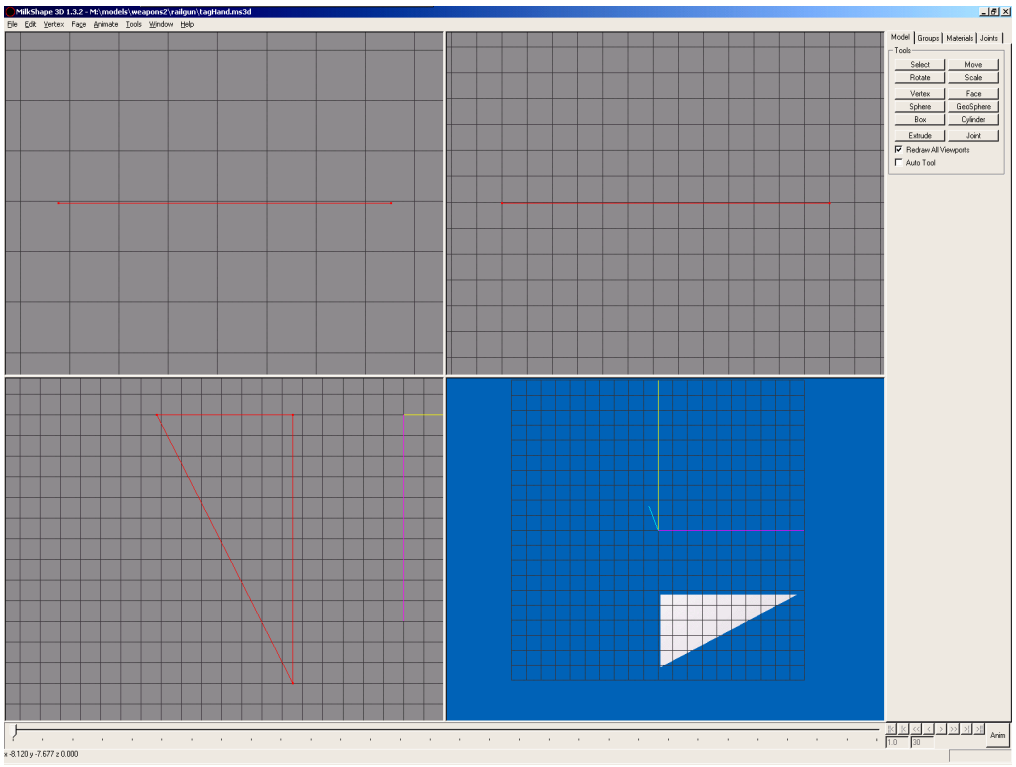

Figure 17. Picture of Model Tag in MilkShape 3D

This tag could then be exported under the filename of any unnecessary weapon. Essentially, this process replaced each unnecessary weapon with an invisible tag. So when Q3A went to render these weapons, it found a tag and placed the weapon as it did before. The engine didn't seem to care that there was no accompanying model to go with

the tag. This left the program running as before, except now the weapons we wanted to eliminate were simply gone.

Our scenario requires that the other bots roam the world looking for our human players to kill. When the mod was first created, the bots stayed in one spot instead of wandering freely. We had to add weapons around the world in order to provide the bots with some impetus for moving. This was a problem for realism though, because now we had a lot of unrealistic weapons lying around. We were forced to render these weapons invisible, so that the players cannot see the weapons that the bots are searching for. By using these weapon tags to make the models invisible, we solved the realism problem and made the bots look natural as they roam the world.

3. Player Skin

Each bot has an underlying skeleton with a skin placed over it. This skin is contained in multiple image files that correspond to different looks that the bot can have. For example, since a bot can be on either a red or blue CTF team, they need red and blue skins. We chose to change biker's skin color to a yellow color in order to make him more recognizable as the hostage. While this may not necessarily make the scenario more realistic, it was also useful for determining how skin changes are made.

Figure 18 shows the skin file for Biker's red uniform before we edited it. We used Adobe Photoshop to edit the skin. There is a feature called color replacement that allows the user to change one color into another. We simply chose the red color and told Photoshop to make it yellow. Figure 19 shows the edited image file. Biker's skin files were then replaced in the Q3A code, giving him a uniform that was different from the other bots

Figure 18. Biker's Skin Before Editing

Figure 19. Biker's Skin After Editing

Once we saw how easy it was to change the skin of a Q3A player, we decided to try something a little more difficult. The goal was to put an actual human into the game on Biker's skeleton. In other words, we wanted to take a sequence of digital photos of a person and make them fit onto Biker's body.

First we took a couple full-length digital photos of a person, along with two photos of his head. These head photos included one face shot and one side shot. Figure 20 shows the image file that is used to texture the bot's head.

Figure 20. Biker's Original Head Texture

It looks as if they peeled the skin off the bot's head and made it lay flat. This posed a problem for us to be able to put our photos into the image. That is why we took two photos of the head. Using PhotoShop, the photo of the face was scaled down and placed over the face of Biker (see Figure 21). The second shot of the side of the head was also scaled down and placed so that the ears lined up (see Figure 21).

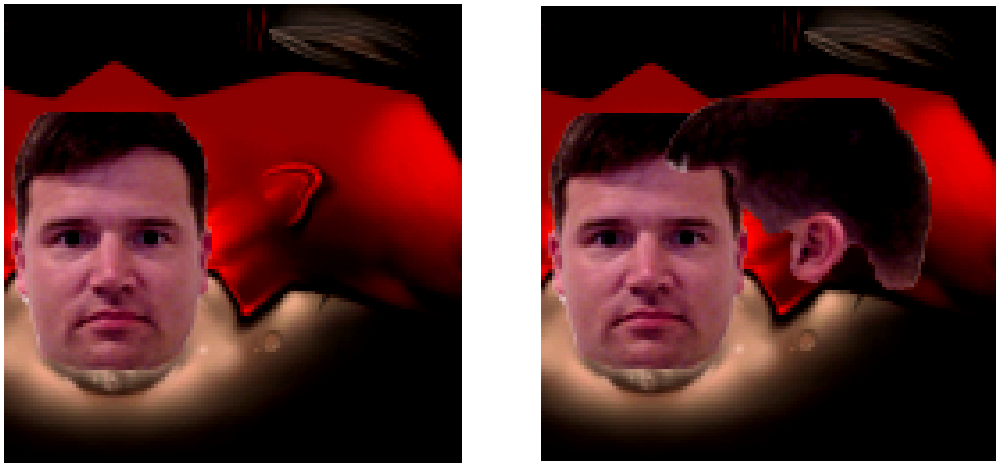

Figure 21. Biker's Head Texture With New Head and Ear Added

The rest of the editing was as much art as science. We used the airbrush feature of PhotoShop which allows the user to copy colors from one place to another. This allowed us to add hair over the entire back and top of Biker's head, along with adding skin to the region between the two added images. Figure 22 shows the final head image with our human replacing Biker.

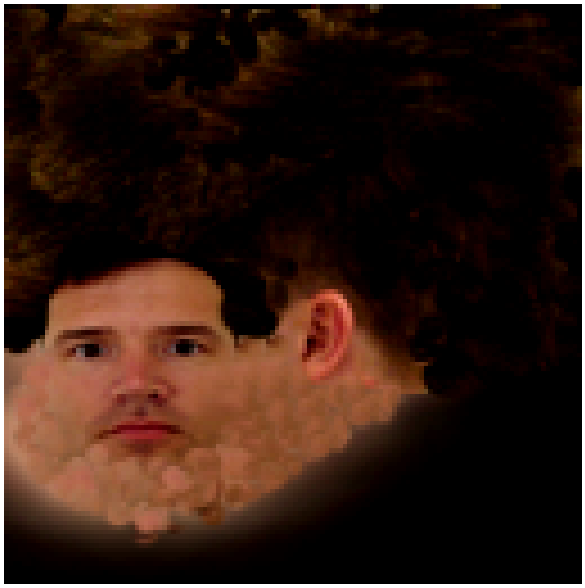

Figure 22. Biker's New Head Texture

Replacing Biker's clothing was much easier than replacing his head texture because the clothing is built in separate pieces. First we took the front-facing digital photo and cut out the shirt. This was placed over the shirt that was in the original image. The rest of the file was completed in much the same way. This image did not require as much artistic ability, since the clothing from our photo matched up well with Biker's original clothing. Figure 23 shows the final image with civilian clothing, which can be compared to the original image in Figure 18. Figure 24 shows the final Biker model with the new skin.

Figure 23. Biker's New Clothing Texture

Figure 24. The New Biker Model in the Game

E. SOURCE CODE CHANGES

To make Assault on Herrmann Hall, several source code changes were needed. In order to provide the reader with examples of changes that might typically be needed if creating a mod, this chapter describes the more extensive changes that were made when creating our mod.

1. Weapons Characteristics (Gauntlet, Railgun)

When the railgun fires, the weapon's effect is similar to a laser light, but it needed some modifications to make it appear more like a laser. There were several steps to give the railgun the characteristics we wanted. First, we removed its capability to cause

damage by changing the code here in Figure 25 (`s_quadFactor` is used to compute if the player has the Quad Damage powerup).

```
// g_weapon.c

//damage = 100 * s_quadFactor;
//radiusDamage = 30 * s_quadFactor;
damage = 0;
radiusDamage = 0;
```

Figure 25. Removing Damage Caused By Railgun

Next, the sounds that the railgun makes had to be muted, using the following code in Figure 26:

```
//cg_weapons.c
//weaponInfo->readySound =
  trap_S_RegisterSound("sound/weapons/railgun/
  rg_hum.wav" );
weaponInfo->readySound =
        trap_S_RegisterSound("sound/misc/silence.wav" );
//weaponInfo->flashSound[0] =
  trap_S_RegisterSound("sound/weapons/railgun/railgf1a.wav" );
weaponInfo->flashSound[0] =
        trap_S_RegisterSound("sound/misc/silence.wav" );
```

Figure 26. Eliminating Railgun Sounds

Finally, to make the effect appear continuous, we increased the firing rate from once every 1500 milliseconds to every millisecond and to make the ammo supply appear infinite (otherwise the light would only work for a very short period of time). Figure 27 contains the code changed to make these changes.

```
// bg_pmove.c
case WP_RAILGUN:
//    addTime = 1500;
      addTime = 1; // "fire" every ms
      break;

pm->ps->ammo[WP_RAILGUN] = 999; // added for the laser light
```

Figure 27. Increasing Railgun Firing Rate

Similar changes were made to the characteristics of the gauntlet. The gauntlet is a weapon that all Q3A characters have at all times. It uses no ammunition, so it is always available. The problem with this is the fact that we wanted Biker, the hostage, to not use any weapons. Since we did not want the human players to use the gauntlet either, the easiest way to solve this problem was to make the gauntlet useless—that is, cause no damage. We made similar changes to the gauntlet code as described above for the railgun, thereby making it useless and silent.

2. Removing Ammunition From Weapons

Each bot in Q3A has its own preference for different weapons. By placing different weapons around the map, we can randomize the paths that each bot takes. (In an early version of the mod, all of the bots would run around the map in a single-file line—not very realistic!) The only problem with using different weapons as "bait" is the fact that we did not want the bots to use those weapons against the players. Therefore, we had to come up with a way to convince the bots to not want to use those weapons once they picked them up. We did this by forcing the ammo count for each of these weapons to zero. Therefore, the bots wouldn't even attempt to use the weapons because they did not have any ammo for them. The bots would still be attracted to the weapons

and try to pick them up, thus randomizing their paths. One other possible way to achieve this effect would have been to change the AI for the bots, but that is more time consuming. Figure 28 shows the source code required to set the ammo count to zero for the weapons.

```
//ai_dmq3.c
//    bs->inventory[INVENTORY_GRENADELAUNCHER] =
//        (bs->cur_ps.stats[STAT_WEAPONS] &
//        (1 << WP_GRENADE_LAUNCHER)) != 0;
//    bs->inventory[INVENTORY_ROCKETLAUNCHER] =
//        (bs->cur_ps.stats[STAT_WEAPONS] &
//        (1 << WP_ROCKET_LAUNCHER)) != 0;
//    bs->inventory[INVENTORY_LIGHTNING] =
//        (bs->cur_ps.stats[STAT_WEAPONS] &
//        (1 << WP_LIGHTNING)) != 0;
//    bs->inventory[INVENTORY_PLASMAGUN] =
//        (bs->cur_ps.stats[STAT_WEAPONS] &
//        (1 << WP_PLASMAGUN)) != 0;
//    bs->inventory[INVENTORY_GRAPPLINGHOOK] =
//        (bs->cur_ps.stats[STAT_WEAPONS] &
//        (1 << WP_GRAPPLING_HOOK)) != 0;
//    bs->inventory[INVENTORY_BFG10K] =
//        (bs->cur_ps.stats[STAT_WEAPONS] &
//        (1 << WP_BFG)) != 0;
      bs->inventory[INVENTORY_GRENADELAUNCHER] = 0;
      bs->inventory[INVENTORY_ROCKETLAUNCHER] = 0;
      bs->inventory[INVENTORY_LIGHTNING] = 0;
      bs->inventory[INVENTORY_PLASMAGUN] = 0;
      bs->inventory[INVENTORY_GRAPPLINGHOOK] = 0;
      bs->inventory[INVENTORY_BFG10K] = 0;
```

Figure 28. Removing Ammunition From Unnecessary Weapons

3. One Life in Gameplay

In normal Q3A gameplay, after a player is killed, they re-spawn with full health, a machine gun with one hundred rounds of ammunition, and the gauntlet. For the purpose of this mod, each player, bot and human, has only have one “life”. To accomplish this, we changed the code so that after a player is killed, they become a game spectator. As a spectator, they can still navigate around the map and observe the game, but cannot

interact with the world or other players. Figure 29 shows the source code required to make a player become a spectator after every death.

```
//g_combat.c
//function player_die
self->client->ps.persistant[ PERS_TEAM ] =
                  TEAM_SPECTATOR;
self->client->sess.sessionTeam = TEAM_SPECTATOR;
self->client->sess.spectatorState = SPECTATOR_FREE;
self->client->ps.pm_flags &= ~PMF_FOLLOW;
self->r.svFlags &= ~SVF_BOT;
self->client->ps.clientNum = ent - g_entities;
SetTeam( self, "spectator" );
self->client->sess.spectatorState =
                  SPECTATOR_FOLLOW;
```

Figure 29. One Life Source Code

4. In-Game Menu Changes

We wanted to create a method for player to rate other players in the game. After each game, a player can use the modified in-game menus to assign a rating to other players. The ratings are based on how well players follow orders and contribute to the team. Figures 30 and 31 show the modified in-game menus. The menus currently do nothing else once a rating has been selected. These menus could easily be implemented to record the rating for later review. The "Rate Player" menu can either be selected by pressing "ESC" which brings up the in-game top level menu and then selecting "Rate Player" as shown in Figure 30, or by pressing the "r" key, which immediately brings up the "Rate Player" menu as shown in Figure 31. Creating the "Rate Player" menu required the addition of a new file in the Quake source code. This file is included in Appendix C.

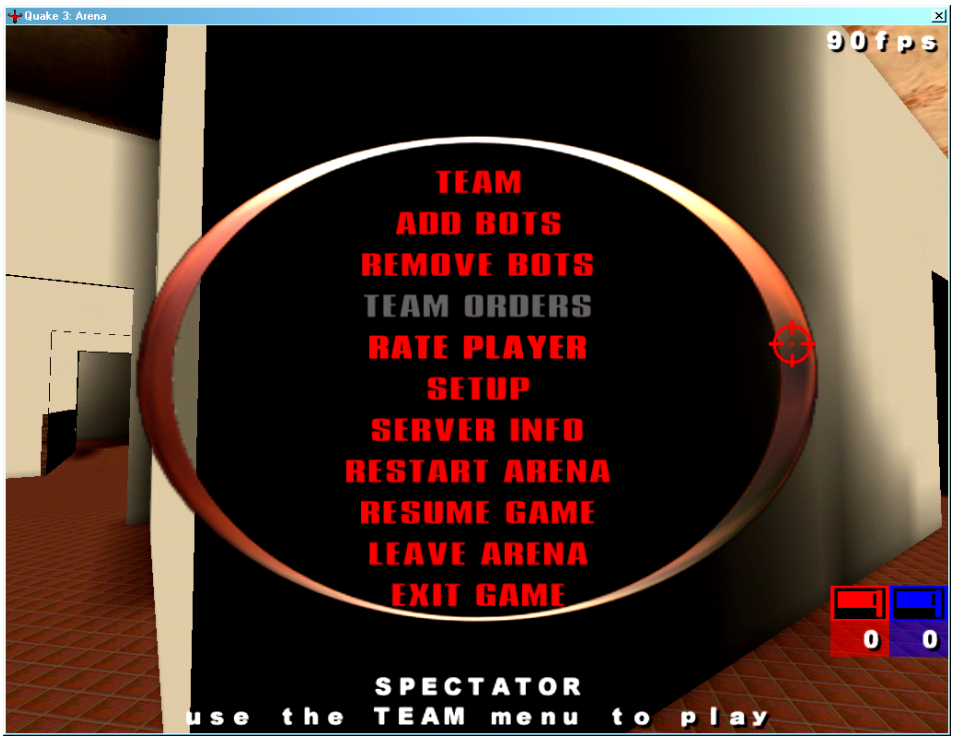

Figure 30. Modified Top Level Menu

Figure 31. Rate Player Menu

F. SUMMARY

This chapter has explored the details of our implementation. By following the steps outlined in Chapter IV, and using the implementation provided in Chapter V, any motivated person should be able to successfully build a modification of their own. While the specifics are unique to our scenario, they should serve as a guide to those who wish to write their own mod. Chapter VI provides an analysis of our modification and explores the benefits of our methodology.

VI. ANALYSIS

This chapter explores the benefits of the modification that we have produced. Included is a summary of the entire body of work, along with the benefits of our work and our conclusions and recommendations. The final section contains ideas for future work.

A. SUMMARY OF WORK

Virtual environment and virtual reality applications are achieving widespread use in the military. Unfortunately, these applications are expensive and have a lengthy development time. We set out to determine if a commercial game could be modified to provide at least the same performance, but at a cheaper cost and with faster development time.

After identifying all of our options, we chose a game to be modified. Next, we developed a scenario that would be the goal of our modification: a team trainer. This mock application could later be made into a training device, but our goal was not to actually incorporate training metrics into the application. Rather, we intended to determine if the modifications were possible, and, if possible, what steps were involved. Once our scenario design was completed, we had to design a level. We chose a real-world building converted from CAD data, since realism was important to our application. After our level was complete, we changed certain models in the game. One change was to incorporate a laser designator for the players to use to point out objectives to other players. Also, all unnecessary weapons were removed from the game to enhance realism. The next step in our modification was changing one of the player's skins to resemble

civilian clothing. This was also an important element for promoting realism. Once all the model changes and player changes were made, we had to change certain elements of the gameplay, such as giving all players only a single life. The last change to be made was to alter the user interface. We added some menu choices that will be useful when team metrics are incorporated into the scenario.

The final product was a proof-of-concept mock application that achieved our goal. We showed that a commercial game could be modified relatively easily and inexpensively, while providing superior graphics and performance. Also, these modifications can themselves be altered to change the application as requirements shift. Commercial game modifications provide a viable alternative to producing applications from scratch for military use.

B. BENEFITS OF REPURPOSING

1. High Fidelity and Fast Performance

Today's game engines are extremely powerful. They are capable of displaying large architectural datasets with very high fidelity, even real-world buildings. Frame rates are well above thirty frames per second on average hardware platforms, which is more than the twenty-four frames per second necessary for visual fusion (Vince, 1992). Writing a modification for a game allows the developer to leverage this technology.

2. Development Time and Cost

Due to this leveraging of technology, mods can be created in a very short amount of time as compared to development from scratch. "Assault on Herrmann Hall" took two graduate students a total of four months to develop (about 300 man hours). These

students were not professional programmers, but they had a basic background in computer programming (professional programmers and artists could undoubtedly create a mod in less time, and with more polished results). The tools to create a mod are readily available, high in quality, easy to use, and often available without cost. These tools are often the same tools that the original game production staff used during the game development. For our mod, the only tools that were not free were Microsoft® Visual Studio® 6.0 for the C++ coding (cost of about $100) and Adobe® PhotoShop® 5.5 (cost of about $600). However, there are excellent freeware substitutes available for these products (gcc and The GIMP). In fact, the Q3A source code is packaged with its own freeware compiler, which we did not use. The only other commercial tool that is commonly used when developing mods (and original games) is 3D Studio Max®. Because we did not modify any animation, we did not require 3D Studio Max®.

With so many tools available for free, the development costs become the major expenses in mod creation. Short development time directly leads to smaller overall salaries for the development staff, keeping the overall cost even lower. Development costs for new games often exceeds one million dollars, but the cost for a basic mod can theoretically be for labor only. Professional mods may not cost much more than a few thousand dollars. Our costs included:

- $6000 for two complete PC workstations
- $100 for Microsoft® Visual Studio® 6.0
- $60 for two licensed copies of Q3A
- $600 for Adobe® PhotoShop® 5.5
- $20 for Milkshape 3D (after a 30-day free trial)

This gave us a total development cost of $6780 (labor costs are not included). This is far less than the cost of developing a VE application from scratch.

C. CONCLUSIONS AND RECOMMENDATIONS

We set out to show that it is possible to modify current games, in a short time, using low cost tools, to produce an application that is useful. Currently, games are often created with the intention of being modified by end users and by professional programmers and artists. However, people with little or no real experience in video game development produce the vast majority of mods. This helps illustrate the fact that simple mods are fairly easy to create. Professional mods are more difficult, but with practice even a novice mod developer can produce a stunning product.

It was easier to create "Assault on Herrmann Hall" as a mod of Q3A than as an original product. Using tools that were inexpensive and easy to use, we completed the mod in approximately four months. In fact, it would have been impossible for us to complete this project as an original program. Without the ability to leverage Q3A's technology, our scenario would have taken at least a year to develop. Even after that year of design and production, our final result could not have approached the quality of a third-generation best-selling game. The fact that id Software only produces games, and they have professional programmers working on those games, means that our mod can take advantage of their expertise, successes and failures. To try and produce a new scenario without leveraging this technology is an inefficient use of resources.

In the process of developing the mod, we evaluated twenty games based on gameplay criteria. This gave us insight into what capabilities current video games have, and also what games have the capability to be modified by the end user. We also

researched the current game engines available, documenting their costs and capabilities. These evaluations provide a mod developer with a place to start when deciding which game to modify for a new application. Our hope is that these evaluations will be stored in a database that can be kept updated for military use. Some kind of central game evaluation repository would be beneficial to those who want to make an informed decision about which game to modify without having to evaluate every game that is currently available.

This thesis has shown that when there is an expressed desire for an application requiring networked interaction between users in real time, serious consideration should be given to modifying current video games to suit that need. Applications that have the best potential for implementation as a game mod are those that require any or all of the following characteristics:

- networking
- extensive use of computer graphics, especially building interiors or barren landscapes
- use of computer-controlled AI characters
- a well-defined world or map that users interact

Another route that may be taken when developing games, or applications similar to games, is to start with a game engine and write an application that uses that engine. This method is considerably more expensive than modifying a current game, but when there are no games similar to the intended application, this may be the only recourse. Choosing this method is still normally faster and less expensive than developing the application from scratch.

As previously stated, modifications allow the developer to leverage the technology currently available in commercial games for a fraction of the cost. In this era of reduced budgets for military simulations and trainers, the modification of commercial games is an attractive alternative to building an application from scratch.

D. FUTURE WORK

While our mock application achieved the goals that were intended, there is more work that can be done to improve understanding of the modification process. In this section we will list some of our suggestions for future work.

1. Implement Training Metrics

One possible area for future work is to modify the source code such that the game actually records leadership data and how well team members follow orders. This type of data would be useful to the military, but would be difficult to implement. No game currently on the market is able to automatically measure team interaction metrics. This also applies to military computer-based trainers.

2. Build a Team Trainer

Our mock application shows that it is possible to make extensive changes to an existing game and create a new application with that game. "Assault on Herrmann Hall" needs additional code changes to be useful as a real team trainer. Teamwork monitoring needs implementing, which fits into the previously mentioned future work of implementing training metrics. Once the trainer is built, a usability study would be an appropriate next step.

3. Build a Theater Planner

We have shown that it is possible to modify a COTS video game to serve as both an architectural walkthrough and as a team trainer. The third implementation of VE/VR software that the military uses is theater planning. It would be very interesting to see if it is possible to modify an existing game to serve as a military theater planner.

4. Build a Game Evaluation Database

As previously mentioned, some kind of central game evaluation repository would be beneficial to those who want to make their own mod. Searching for the optimal game to modify for a certain application would be easier with all evaluations stored in one place and updated as new games are produced.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A.CONVERTER CODE

```
// dxf2map.cpp : Defines the entry point for the console application.
//
// Skip Morrow
// Jeff DeBrine
// 15 Aug 2000
//
// Converts an AutoCAD dxf file to a map file that can
// be read by worldcraft.
//

#include "stdafx.h"
#include <stdlib.h>
#include <iostream>
#include <iomanip.h>
#include <fstream.h>
#include <math.h>
#include <string>
#include <vector>

//using namespace std;

struct coordinate{
    float fX, fY, fZ;
};

// holds four coordinates
struct fourtuple {
    coordinate coord1, coord2, coord3, coord4;
};

// this subtracts two coordinates
coordinate subtract(coordinate c1, coordinate c2)
{
    coordinate tCoord;
    tCoord.fX = c1.fX - c2.fX;
    tCoord.fY = c1.fY - c2.fY;
    tCoord.fZ = c1.fZ - c2.fZ;
    return (tCoord);
}

// calculates the cross product
coordinate cross(coordinate c1, coordinate c2)
{
    coordinate tCoord;
    float fMag;
    tCoord.fX = c1.fY * c2.fZ - c1.fZ * c2.fY;
    tCoord.fY = -(c1.fX * c2.fZ - c1.fZ * c2.fX);
    tCoord.fZ = c1.fX * c2.fY - c2.fX * c1.fY;

    fMag = sqrt(tCoord.fX * tCoord.fX +
                tCoord.fY * tCoord.fY +
                tCoord.fZ * tCoord.fZ);

    tCoord.fX = tCoord.fX / fMag;
```

```
    tCoord.fY = tCoord.fY / fMag;
    tCoord.fZ = tCoord.fZ / fMag;
    return (tCoord);
}

// calculates the dot product of two coordinates
float dot(coordinate c1, coordinate c2)
{
    return(c1.fX * c2.fX + c1.fY * c2.fY + c1.fZ * c2.fZ);
}

// Face coords must be written clockwise as seen from the outside of
// the brush.  To ensure that they are written clockwise, choose a
// point from the oposite side of the brush as the viewing point.  This
// point could just as effectively be *inside* the brush, so I know
// that the points should appear counter-clockwise from this point.  To
// find out if three points in 3d space are clockwise or
// counterclockwise, first choose a vertex among the three points.
// Subtract the vertex from the other two points, and you have two
// vectors with their origins at the vertex point. Next subract the
// vertex from the view point, and you have a vectr from the vertex to
// the view point.  Take the cross product of the first two vectors,
// and then the dot product of that cross product and the vector to the
// viewpoint.  If the dot product is negative, the points are clockwise
// as viewed from the outside of the brush.  Write the points to the
// file in the correct order based on the sign (+/-) of the dot product

void writeClockwise(coordinate c1, coordinate c2, coordinate c3,
                                                coordinate vis, ofstream &
ofile)
{
    coordinate v1, v2, vCross, vDot;
    float dProd;
    v1 = subtract(c2, c1);
    v2 = subtract(c3, c1);
    vCross = cross(v1, v2);
    vDot = subtract(vis, c1);
    dProd = dot(vCross, vDot);

    if (dProd > 0) // write the coords in this order
    {
      ofile << "( " << c1.fX << " " << c1.fY << " " << c1.fZ << " ) ( "
                  << c2.fX << " " << c2.fY << " " << c2.fZ << " ) ( "
                  << c3.fX << " " << c3.fY << " "<< c3.fZ << " ) "
                  << "base_wall/concrete 0 0 0 1.0 1.0 0 0 0" << endl;
    }
    else // or this order, depending on the sign of the dot product
    {
     ofile << "( " << c1.fX << " " << c1.fY << " " << c1.fZ << " ) ( "
                    << c3.fX << " " << c3.fY << " " << c3.fZ << " ) ("
                    << c2.fX << " " << c2.fY << " " << c2.fZ << " ) "
                    << "base_wall/concrete 0 0 0 1.0 1.0 0 0 0" << endl;
    }
}

// Takes a plane, extrudes it 2 units along its surface normal
// and makes it a solid by calculating the other five planes
```

```
void makeSolid(fourtuple ft, ofstream & of)
{
    coordinate v1, v2, v;
    fourtuple bf; // backface
    float fExtrude = 2.0; // distance to extrude the plane

    // first get two vectors on the plane
    v1 = subtract(ft.coord1, ft.coord2);
    v2 = subtract(ft.coord1, ft.coord3);

    v = cross(v1, v2); // get the surface normal


    // calculate the coordinates of all the verticies on the
    // plane on the oposite side (four more points in 3d space)
    bf.coord1.fX = ft.coord1.fX + v.fX * fExtrude;
    bf.coord1.fY = ft.coord1.fY + v.fY * fExtrude;
    bf.coord1.fZ = ft.coord1.fZ + v.fZ * fExtrude;

    bf.coord2.fX = ft.coord2.fX + v.fX * fExtrude;
    bf.coord2.fY = ft.coord2.fY + v.fY * fExtrude;
    bf.coord2.fZ = ft.coord2.fZ + v.fZ * fExtrude;

    bf.coord3.fX = ft.coord3.fX + v.fX * fExtrude;
    bf.coord3.fY = ft.coord3.fY + v.fY * fExtrude;
    bf.coord3.fZ = ft.coord3.fZ + v.fZ * fExtrude;

    bf.coord4.fX = ft.coord4.fX + v.fX * fExtrude;
    bf.coord4.fY = ft.coord4.fY + v.fY * fExtrude;
    bf.coord4.fZ = ft.coord4.fZ + v.fZ * fExtrude;


    // write all of the coordinates - must be clockwise!
    of << "{" << endl;
    //front
    writeClockwise(ft.coord1, ft.coord2, ft.coord3, bf.coord1, of);
    //back
    writeClockwise(bf.coord1, bf.coord2, bf.coord3, ft.coord1, of);
    //top
    writeClockwise(bf.coord1, bf.coord2, ft.coord2, bf.coord4, of);
    //bottom
    writeClockwise(ft.coord4, ft.coord3, bf.coord3, ft.coord1, of);
    //left
    writeClockwise(ft.coord2, bf.coord2, bf.coord3, ft.coord1, of);
    //right
    writeClockwise(bf.coord1, ft.coord1, ft.coord4, bf.coord2, of);
    of << "}" << endl;
}



int main(int argc, char* argv[])
{
    ifstream inFile;
    ofstream outFile;

    char cGroupCode[255];
```

```
char cValue[255];
int iGroupCode;
float fValue;
float fScaleX = 32.0;
float fScaleY = 32.0;
float fScaleZ = 32.0;
cValue[0] = NULL;
int iCoordCount = 0;
int iFaceCount = 0;
long int iLineNumber = 0;
int iBrushCount = 0;

coordinate tempCoordinate;
fourtuple tempfourtuple;


// open the input file
if (!argv[1])
{
    argv[1] = "inFile.dxf";  // default file name
} // end if


cout << "Opening " << argv[1] << "." << endl;
inFile.open(argv[1], ios::nocreate);

if (!inFile)
{
    cout << "Error!  Could not open " << argv[1] << endl;
} // end if


// open the output file
if (!argv[2])
{
    argv[2] = "outFile.map";  // default file name
} // end if

cout << "Creating " << argv[2] << "." << endl;
outFile.open(argv[2]);

if (!outFile)
{
    cout << "Error!  Could not create " << argv[2] << endl;
} // end if


// write the worldspawn entry at the top
// note that the last outfile is one long line.  Change this line
// as needed to match the correct worldspawn that you need.
outFile << "{" << endl;
outFile << "\"classname\" \"worldspawn\"" << endl;
outFile << "\"wad\" \"\\sierra\\half-life\\valve\\halflife.wad;"
        << "\\sierra\\half-life\\valve\\liquids.wad;"
        << "\\sierra\\half-life\\valve\\xeno.wad;"
        << "\\sierra\\half-life\\valve\\decals.wad\"" << endl;
```

```
    // First, find the ENTITIES section (where the geometry is stored)
    // in the dxf file.  There's nothing before this line that I need
    bool bFound = false;
    while (!bFound)
    {
        inFile.getline(cGroupCode, 254);
        iLineNumber++;
        inFile.getline(cValue, 254);
        iLineNumber++;
        iGroupCode = atoi(cGroupCode);

        if ((iGroupCode == 2) && (!strcmp(cValue, "ENTITIES")))
        {
            cout << "Entities Section Found" << endl;
            bFound = true;
        }
    }

    cout << setiosflags(ios::fixed) << setprecision(2);
    outFile << setiosflags(ios::fixed) << setprecision(0);

    // Now, start looking for geometry
    while (inFile.getline(cGroupCode, 254))
    {
        iLineNumber++;
        inFile.getline(cValue, 254);
        iLineNumber++;
        iGroupCode = atoi(cGroupCode);
        fValue = atof(cValue);

        if ((iGroupCode == 0) && ((!strcmp(cValue, "3DFACE")) ||
                                  (!strcmp(cValue, "POLYLINE"))))
        {
//          cout << "3DFACE found" << endl;
            iCoordCount = 0;
        }

        // it is a coordinate for a 3DFACE
        if ((iGroupCode > 9) && (iGroupCode < 40) && (iCoordCount <
                                                                13))
        {
            iCoordCount++;

            // "x" coords have a group code between 10 and 20
            if (iGroupCode < 20)
            {
                tempCoordinate.fX = fValue * fScaleX;
            }

            // "z" coords have a group code between 30 and 40
            if (iGroupCode > 29)
            {
                tempCoordinate.fZ = fValue * fScaleZ;
            }

            // "y" coords have a group code between 20 and 30
```

```
            if ((iGroupCode > 19) && (iGroupCode < 30))
            {
                tempCoordinate.fY = fValue * fScaleY;
            }

            if (iCoordCount == 3)
            {
                tempfourtuple.coord1=tempCoordinate;
            }

            if (iCoordCount == 6)
            {
                tempfourtuple.coord2=tempCoordinate;
            }

            if (iCoordCount == 9)
            {
                tempfourtuple.coord3=tempCoordinate;
            }

            // I now have four coordinates
            if (iCoordCount == 12)
            {
                iCoordCount = 0;
                tempfourtuple.coord4=tempCoordinate;

           // Definitely a hack.  I was getting some strange brushes
           // because sometimes the first two coordinates were the same
           // or I was getting some really high z values.  I chose to
           // just not write those coordinates.  Out of a 20,000
           // polygon model, such as Herrmann Hall, I only lost about
           // 50 polygons this way.  I don't know if it was because of
           // a problem in this code, or if it was an artifact in the
           // dxf file.
/*              if (!((tempfourtuple.coord1.fX ==
                                        tempfourtuple.coord2.fX) &&
                    (tempfourtuple.coord1.fY ==
                                        tempfourtuple.coord2.fY) &&
                    (tempfourtuple.coord1.fZ ==
                                       tempfourtuple.coord2.fZ)) &&
                    (tempfourtuple.coord1.fZ < 200) &&
                    (tempfourtuple.coord2.fZ < 200) &&
                    (tempfourtuple.coord3.fZ < 200) &&
                    (tempfourtuple.coord4.fZ < 200)) */
                if (!((tempfourtuple.coord1.fX ==
                                        tempfourtuple.coord2.fX) &&
                    (tempfourtuple.coord1.fY ==
                                        tempfourtuple.coord2.fY) &&
                    (tempfourtuple.coord1.fZ ==
                                        tempfourtuple.coord2.fZ)))
                {
                    outFile << "// brush " << iBrushCount << "\n";
                    iBrushCount++;
                    makeSolid(tempfourtuple, outFile);
                }
            }
        }
```

```
    }
    cout << "Done reading input file\n";
    outFile << "}" << endl;

    inFile.close();

    return 0;
}
// eof dxf2map
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. Q3A ARTIFICIAL INTELLIGENCE CODE

```
//===========================================================================
// ***MODIFIED CODE***
// Name:                Biker_c.c
// Function:            Biker, rank 2
// Programmer:          MrElusive (MrElusive@idsoftware.com)
// version:             1
// Tab Size:            4 (real tabs)
//===========================================================================

#include "chars.h"

skill 1
{
      CHARACTERISTIC_NAME                           "Biker"
      CHARACTERISTIC_GENDER                         "male"
      CHARACTERISTIC_ATTACK_SKILL                   0.0
      CHARACTERISTIC_WEAPONWEIGHTS                  "bots/biker_w.c"
      CHARACTERISTIC_AIM_SKILL                      0.0
      CHARACTERISTIC_AIM_ACCURACY                   0.0
      CHARACTERISTIC_VIEW_FACTOR                    0.0
      CHARACTERISTIC_VIEW_MAXCHANGE                 180
      CHARACTERISTIC_REACTIONTIME                   5.0

      CHARACTERISTIC_CHAT_FILE                      "bots/biker_t.c"
      CHARACTERISTIC_CHAT_NAME                      "biker"
      CHARACTERISTIC_CHAT_CPM                       400
      CHARACTERISTIC_CHAT_INSULT                    0.0
      CHARACTERISTIC_CHAT_MISC                      0.0
      CHARACTERISTIC_CHAT_STARTENDLEVEL             0.5
      CHARACTERISTIC_CHAT_ENTEREXITGAME             0.5
      CHARACTERISTIC_CHAT_KILL                      0.0
      CHARACTERISTIC_CHAT_DEATH                     0.0
      CHARACTERISTIC_CHAT_ENEMYSUICIDE              0.0
      CHARACTERISTIC_CHAT_HITTALKING                0.0
      CHARACTERISTIC_CHAT_HITNODEATH                0.0
      CHARACTERISTIC_CHAT_HITNOKILL                 0.0
      CHARACTERISTIC_CHAT_RANDOM                    0.0
      CHARACTERISTIC_CHAT_REPLY                     0.0

      CHARACTERISTIC_CROUCHER                       0.0
      CHARACTERISTIC_JUMPER                         0.0
      CHARACTERISTIC_WEAPONJUMPING                  0.0
      CHARACTERISTIC_GRAPPLE_USER                   0.0

      CHARACTERISTIC_ITEMWEIGHTS
      "bots/biker_i.c"
      CHARACTERISTIC_AGGRESSION                     0.0
      CHARACTERISTIC_SELFPRESERVATION               1.0
      CHARACTERISTIC_VENGEFULNESS                   0.0
      CHARACTERISTIC_CAMPER                         0.0

      CHARACTERISTIC_EASY_FRAGGER                   0.0
      CHARACTERISTIC_ALERTNESS                      0.0
}
```

```
skill 4
{
	CHARACTERISTIC_NAME				"Biker"
	CHARACTERISTIC_GENDER				"male"
	CHARACTERISTIC_ATTACK_SKILL			0.0
	CHARACTERISTIC_WEAPONWEIGHTS			"bots/biker_w.c"
	CHARACTERISTIC_AIM_SKILL			0.0
	CHARACTERISTIC_AIM_ACCURACY			0.0
	CHARACTERISTIC_VIEW_FACTOR			0.0
	CHARACTERISTIC_VIEW_MAXCHANGE			180
	CHARACTERISTIC_REACTIONTIME			5.0

	CHARACTERISTIC_CHAT_FILE			"bots/biker_t.c"
	CHARACTERISTIC_CHAT_NAME			"biker"
	CHARACTERISTIC_CHAT_CPM				400
	CHARACTERISTIC_CHAT_INSULT			0.0
	CHARACTERISTIC_CHAT_MISC			0.0
	CHARACTERISTIC_CHAT_STARTENDLEVEL		0.5
	CHARACTERISTIC_CHAT_ENTEREXITGAME		0.5
	CHARACTERISTIC_CHAT_KILL			0.0
	CHARACTERISTIC_CHAT_DEATH			0.0
	CHARACTERISTIC_CHAT_ENEMYSUICIDE		0.0
	CHARACTERISTIC_CHAT_HITTALKING			0.0
	CHARACTERISTIC_CHAT_HITNODEATH			0.0
	CHARACTERISTIC_CHAT_HITNOKILL			0.0
	CHARACTERISTIC_CHAT_RANDOM			0.0
	CHARACTERISTIC_CHAT_REPLY			0.0

	CHARACTERISTIC_CROUCHER				0.0
	CHARACTERISTIC_JUMPER				0.0
	CHARACTERISTIC_WEAPONJUMPING			0.0
	CHARACTERISTIC_GRAPPLE_USER			0.0

	CHARACTERISTIC_ITEMWEIGHTS			"bots/biker_i.c"
	CHARACTERISTIC_AGGRESSION			0.0
	CHARACTERISTIC_SELFPRESERVATION			1.0
	CHARACTERISTIC_VENGEFULNESS			0.0
	CHARACTERISTIC_CAMPER				0.0

	CHARACTERISTIC_EASY_FRAGGER			0.0
	CHARACTERISTIC_ALERTNESS			0.0
}

skill 5
{
	CHARACTERISTIC_NAME				"Biker"
	CHARACTERISTIC_GENDER				"male"
	CHARACTERISTIC_ATTACK_SKILL			0.0
	CHARACTERISTIC_WEAPONWEIGHTS			"bots/biker_w.c"
	CHARACTERISTIC_AIM_SKILL			0.0
	CHARACTERISTIC_AIM_ACCURACY			0.0
	CHARACTERISTIC_VIEW_FACTOR			0.0
	CHARACTERISTIC_VIEW_MAXCHANGE			240
	CHARACTERISTIC_REACTIONTIME			5.0

	CHARACTERISTIC_CHAT_FILE			"bots/biker_t.c"
```

```
	CHARACTERISTIC_CHAT_NAME                        "biker"
	CHARACTERISTIC_CHAT_CPM                         400
	CHARACTERISTIC_CHAT_INSULT                      0.0
	CHARACTERISTIC_CHAT_MISC                        0.0
	CHARACTERISTIC_CHAT_STARTENDLEVEL               0.5
	CHARACTERISTIC_CHAT_ENTEREXITGAME               0.5
	CHARACTERISTIC_CHAT_KILL                        0.0
	CHARACTERISTIC_CHAT_DEATH                       0.0
	CHARACTERISTIC_CHAT_ENEMYSUICIDE                0.0
	CHARACTERISTIC_CHAT_HITTALKING                  0.0
	CHARACTERISTIC_CHAT_HITNODEATH                  0.0
	CHARACTERISTIC_CHAT_HITNOKILL                   0.0
	CHARACTERISTIC_CHAT_RANDOM                      0.0
	CHARACTERISTIC_CHAT_REPLY                       0.0

	CHARACTERISTIC_CROUCHER                         0.0
	CHARACTERISTIC_JUMPER                           0.0
	CHARACTERISTIC_WEAPONJUMPING                    0.0
	CHARACTERISTIC_GRAPPLE_USER                     0.0

	CHARACTERISTIC_ITEMWEIGHTS                      "bots/biker_i.c"
	CHARACTERISTIC_AGGRESSION                       0.0
	CHARACTERISTIC_SELFPRESERVATION                 1.0
	CHARACTERISTIC_VENGEFULNESS                     0.0
	CHARACTERISTIC_CAMPER                           0.0

	CHARACTERISTIC_EASY_FRAGGER                     0.0
	CHARACTERISTIC_ALERTNESS                        0.0
}
```

```
//===========================================================================
// ***MODIFIED CODE***
// Name:                visor_i.c
// Function:
// Programmer:          Mr Elusive (MrElusive@idsoftware.com)
// Last update:         1999-09-08
// Tab Size:            4 (real tabs)
//===========================================================================

#include "inv.h"

//initial health/armor states
#define FS_HEALTH                                       1
#define FS_ARMOR                                        1

//initial weapon weights
#define W_SHOTGUN                                       1
#define W_MACHINEGUN                                    1
#define W_GRENADELAUNCHER                               1
#define W_ROCKETLAUNCHER                                1
#define W_RAILGUN                                       1
#define W_BFG10K                                        1
#define W_LIGHTNING                                     1
#define W_PLASMAGUN                                     1

//the bot has the weapons, so the weights change a little bit
#define GWW_SHOTGUN                                     1
#define GWW_MACHINEGUN                                  1
#define GWW_GRENADELAUNCHER                             1
#define GWW_ROCKETLAUNCHER                              1
#define GWW_RAILGUN                                     1
#define GWW_BFG10K                                      1
#define GWW_LIGHTNING                                   1
#define GWW_PLASMAGUN                                   1

//initial powerup weights
#define W_TELEPORTER                                    1
#define W_MEDKIT                                        1
#define W_QUAD                                          1
#define W_ENVIRO                                        1
#define W_HASTE                                         1
#define W_INVISIBILITY                                  1
#define W_REGEN                                         1
#define W_FLIGHT                                        1

//flag weight
#define FLAG_WEIGHT                                     1

//
#include "fw_items.c"
```

```
//==========================================================================
// ***MODIFIED CODE***
// Name:                Biker_c.c
// Function:            chat lines for Biker
// Programmer:          MrElusive (MrElusive@idsoftware.com)
// Author:              The Seven Swords & R.A. Salvatore
// Editor:              Paul Jaquays
// Last update:         Oct. 10, 1999
// Tab Size:            3 (real tabs)
//==========================================================================

//example initial chats
chat "biker"
{
      //the teamplay.h file is included for all kinds of teamplay chats
      #include "teamplay.h"

      //========================================================
      //========================================================

      type "game_enter" //initiated when the bot enters the game
      {
            HELLO5;
            "I am a great hostage.";
/*          "Make this a good day, cuz it'll be your last.";
            "Everyone out of the pool.";
            "You just keep your hands off my hawg, ", 1, ".
Understand?";
            1, ", I'm gonna kick your sorry butt into the next
county.";
            "I squash cockroaches bigger and scarier than you, ", 1,
".";
            "This better not make me late for lunch.";  */
            // 0 = bot name
      } //end type

      type "game_exit" //initiated when the bot exits the game
      {
            "I have some boredom to catch up on.";
            "Screw this.";
            GOODBYE1;
            // 0 = bot name
      } //end type

      type "level_start" //initiated when a new level starts
      {
            LEVEL_START1;
            "I'm makin' ", 4, " my playground. Get off the swings, ",
fighter, "s.";
            "Where'd I park my hawg?";
            // 0 = bot name
            // 1 = randomly chosen player
            // 4 = Level's title
            } //end type

      type "level_end" //initiated when a level ends and the bot is not
first and not last in the rankings
```

```
      {
            "Waste of time.";
            LEVEL_END0;
            2, " cheats and ", 3, " sucks. Now that we know, let's go
again.";
            // 0 = bot name
            // 2 = opponent in first place
            // 3 = opponent in last place
      } //end type

      type "level_end_victory" //initiated when a level ends and the
bot is first in the rankings
      {
            "Yeehaa! Kick some doggie butt!";
            "Did that hurt ya, ", 3, "?  Bet it did!";
            LEVEL_END_VICTORY3;
            // 0 = bot name
            // 3 = opponent in last place
      } //end type

      type "level_end_lose" //initiated when a level ends and the bot
is last in the rankings
      {
            "Aw, you can kiss my bike wheel ", 2, " .... but lemme get
it spinnin' real fast first.";
            "That's it! I'm putting me a rifle rack on my hawg.";

            LEVEL_END_LOSE1;
            // 0 = bot name
      } //end type

      //=======================================================
      //=======================================================

      type "hit_talking" //bot is hit while chat balloon is visible;
lecture attacker on poor sportsmanship
      {
            //"You just became road-kill, ", fighter, ".";
            0, ", you just became a kick-stand.";
            "Nice ~one, ", 0, ". But now it's my turn.";
            //0 = shooter
      } //end type

      type "hit_nodeath" //bot is hit by an opponent's weapon attack;
either praise or insult
      {
            "C'mon, dawg, I still gots ~one good eye!";
            "Don't dirty the leathers, ", 0,"!";
            "You cow-poker! I'll show you what's what!";
            "OK, you gave it your best and now it's my turn.";
            "I'm gonna hurt you, ", 0, ". I'm gonna hurt you bad.";
            DEATH_INSULT2;
            //0 = shooter
      } //end type

      type "hit_nokill" //bot hits an opponent but does not kill it
      {
```

```
            TAUNT1;
            "Yer as slow as ya are ugly.";
            TAUNT8;
            "You look mighty pretty in blood red.";
            "Lousy, cheap ammo.";
            "Sheee-oooot! I gotta get me a bigger gun than this ~one.";
            //0 = opponent
      } //end type

      //=========================================================
      //=========================================================

      type "death_telefrag" //initiated when the bot is killed by a
telefrag
      {
            "Them trucks never give a hawg-rider no respect.";
            "That was some serious road rash.";
            "Anybody get the license plate on that elephant?";
            DEATH_TELEFRAGGED0;
            // 0 = enemy name
      } //end type

      type "death_cratered" //initiated when the bot is killed by
falling damage
      {
            "Them Kneivel boys ain't got nothin' on me.";
            "May be somethin' to them helmet laws ...";
            "Dang! Colorado Rocky Mountain high!";
            // 0 = enemy name
      } //end type

      type "death_lava" //initiated when the bot dies in lava
      {
            "Dang! That's hotter than a tailpipe runnin' jet fuel.";
            "Hoooooweeee! Too much lighter fluid on the grill there.";
            DEATH_LAVA1;
            // 0 = enemy name
      } //end type

      type "death_slime" //initiated when the bot dies in slime
      {
            "Any biscuits with this gravy?";
            "This is like showerin' in degreaser.";
            "Hey Lucy, this stuff's better'n yer maw's cookin'!";
            // 0 = enemy name
      } //end type

      type "death_drown" //initiated when the bot drowns
      {
            "Leather's heavy when it's wet. Dang.";
            "Great, now I smells like a wet dog.";
            "Stupid place for a swimmmin' pool.";
            // 0 = enemy name
      } //end type

      type "death_suicide" //initiated when bot blows self up with a
weapon or craters
```

```
	{
		"You so much as giggle, ", 0, ", and I break yer face.";
		"Did that truck run over my bike too?";
		DEATH_SUICIDE0;
		DEATH_SUICIDE5;
		// 0 = enemy name
	} //end type

	type "death_gauntlet" //initiated when the bot is killed by a
	gauntlet attack
	{
		"Get yer slimy paws off o' me, ", 0, "!";
		"Last guy that touched me can only count to ~five now.";
		"Ain't nobody touches me like that.";
		DEATH_GAUNTLET2;
		// 0 = enemy name
	} //end type

	type "death_rail" //initiated when the bot is killed by a rail
gun shot
	{
		DEATH_RAIL2;
		"This ain't no campground, ", 0, ".";
		"Let's settle this outside, face to face.";
		// 0 = enemy name
	} //end type

	type "death_bfg" //initiated when the bot died by a BFG
	{
		"And the friggin' rockets red ... uhhh ... green glare!";
		DEATH_BFG2;
		"Ha, ya missed, ", 0, "! 'Almost' only counts in hand
grenades and ... oh, puke ...";
		"Stampede!";
		// 0 = enemy name
	} //end type

	type "death_insult" //insult initiated when the bot died
	{
		"I take my coffee stronger'n you, ", 0, ".";
		"You just stepped on the highway to hell, ", fighter, ".";
		"I can't believe a ", fighter, " like ", 0, " fragged me!";
		DEATH_INSULT0;
		DEATH_INSULT4;
		DEATH_INSULT5;
		// 0 = enemy name
	} //end type

	type "death_praise" //praise initiated when the bot died
	{
		"Yeah, big hairy deal, ", 0, ". Ya killed me. So what?";
		"Well, ain't ", 0, " a ~hero?  Come and get your special
prize.";
		D_PRAISE2;
		0, " killed me to death! Dang!";
		// 0 = enemy name
	} //end type
```

```
	//=======================================================
	//=======================================================

	type "kill_rail" //initiated when the bot kills someone with rail
gun
	{
		"Turn into the wind, ", fighter, ", I wanna hear ya
whistle.";
		KILL_RAIL1;
		KILL_RAIL0;
		// 0 = enemy name
	} //end type

	type "kill_gauntlet" //initiated when the bot kills someone with
gauntlet
	{
		"Up close and impersonal.";
		"Like swattin' flies offa roadkill.";
		"That's the kind of action I like most.";
		"Yeah! Gimme some more of that!";
		KILL_GAUNTLET0;
		// 0 = enemy name
	} //end type

	type "kill_telefrag" //initiated when the bot telefragged someone
	{
		"Here's the beef!";
		"Get off the Road!";
		"Popped you like a Prairie Oyster!";
		TELEFRAGGED0;
		// 0 = enemy name
	} //end type

	type "kill_suicide" //initiated when the player kills self
	{
		"Hey moron, AIM, then fire!";
		"Must tickle, cuz I'm laughin'";
		// 0 = enemy name
	} //end type

	type "kill_insult" //Insult initiated when the bot killed someone
	{
		"I'm gonna bury ", 0, " face down so I got a place to park
my bike.";
		"Ahhhh, you ain't worth the trouble to run you over.";
		"Biker 1, Roadkill 0.";
		KILL_INSULT0;
		KILL_INSULT4;
		KILL_INSULT8;
		// 0 = enemy name
	} //end type

	type "kill_praise" //praise initiated when the bot killed someone
	{
		KILL_INSULT36;
		"You'll make good maggot food, ", 0, ". That's something.";
```

```
            "You was brain-dead anyway, ", 0, ".";

            // 0 = enemy name
      } //end type

      //=======================================================
      //=======================================================

      type "random_insult" //insult initiated randomly (just when the
bot feels like it)
      {
            TAUNT7;
            MISC6;
            "All the sex appeal of open heart surgery!";
            "Yer like sand on the road, ", 0, ".I just blow it away.";
            "Your mamma's gonna see you on a milk carton, ", 0, ".";
            "You stay away from my dawg, ", 0, ".";
            // 0 = name of randomly chosen player
            // 1 = bot name
      } //end type

      type "random_misc" //miscellanous chats initiated randomly
      {
            GUYTALK2;
            "I eat my road kill.";
            "*[Bu-u-u-u-u-u-urp]* ";
            "Anybody got a ~six-pack on 'em?";
            "Okay, any o' you pretty ladies want a ride on my hawg?";
            one_liners;
            "Don't go tellin' Lucy I was here ... she'll whip my
butt.";
            femalebot, " better git her ass back in the kitchen and
finish making me that chicken pot pie!";
            MISC0;
            MISC8;
            MISC12;
            // 0 = name of randomly chosen player
            // 1 = bot name
      } //end type
} //end biker chat
```

```
//=====================================================================
// ***MODIFIED CODE***
// Name:                visor_w.c
// Function:
// Programmer:
// Last update:
// Tab Size:            4 (real tabs)
//=====================================================================

#include "inv.h"

#define W_GAUNTLET                                      1
#define W_SHOTGUN                                       1
#define W_MACHINEGUN                                    1
#define W_GRENADELAUNCHER                               1
#define W_ROCKETLAUNCHER                                1
#define W_RAILGUN                                       1
#define W_BFG10K                                        1
#define W_LIGHTNING                                     1
#define W_PLASMAGUN                                     1
#define W_GRAPPLE                                       1

//
#include "fw_weap.c"
```

```
//===========================================================================
// ***ORIGINAL CODE***
// Name:                Biker_c.c
// Function:            Biker, rank 2
// Programmer:          MrElusive (MrElusive@idsoftware.com)
// version:             1
// Tab Size:            4 (real tabs)
//===========================================================================

#include "chars.h"

skill 1
{
      CHARACTERISTIC_NAME
      "Biker"
      CHARACTERISTIC_GENDER                           "male"
      CHARACTERISTIC_ATTACK_SKILL                     0.4
      CHARACTERISTIC_WEAPONWEIGHTS                    "bots/biker_w.c"
      CHARACTERISTIC_AIM_SKILL                        0.5
      CHARACTERISTIC_AIM_ACCURACY                     0.5
      CHARACTERISTIC_VIEW_FACTOR                      0.5
      CHARACTERISTIC_VIEW_MAXCHANGE                   180
      CHARACTERISTIC_REACTIONTIME                     2.5

      CHARACTERISTIC_CHAT_FILE                        "bots/biker_t.c"
      CHARACTERISTIC_CHAT_NAME                        "biker"
      CHARACTERISTIC_CHAT_CPM                         400
      CHARACTERISTIC_CHAT_INSULT                      0.95
      CHARACTERISTIC_CHAT_MISC                        0.5
      CHARACTERISTIC_CHAT_STARTENDLEVEL               0.5
      CHARACTERISTIC_CHAT_ENTEREXITGAME               0.5
      CHARACTERISTIC_CHAT_KILL                        0.5
      CHARACTERISTIC_CHAT_DEATH                       0.5
      CHARACTERISTIC_CHAT_ENEMYSUICIDE                0.5
      CHARACTERISTIC_CHAT_HITTALKING                  0.15
      CHARACTERISTIC_CHAT_HITNODEATH                  0.5
      CHARACTERISTIC_CHAT_HITNOKILL                   0.5
      CHARACTERISTIC_CHAT_RANDOM                      0.5
      CHARACTERISTIC_CHAT_REPLY                       0.4

      CHARACTERISTIC_CROUCHER                         0.25
      CHARACTERISTIC_JUMPER                           0.25
      CHARACTERISTIC_WEAPONJUMPING                    0.5
      CHARACTERISTIC_GRAPPLE_USER                     0.5

      CHARACTERISTIC_ITEMWEIGHTS                      "bots/biker_i.c"
      CHARACTERISTIC_AGGRESSION                       0.75
      CHARACTERISTIC_SELFPRESERVATION                 0.5
      CHARACTERISTIC_VENGEFULNESS                     0.5
      CHARACTERISTIC_CAMPER                           0.5

      CHARACTERISTIC_EASY_FRAGGER                     0.5
      CHARACTERISTIC_ALERTNESS                        0.5
}
```

```
skill 4
{
	CHARACTERISTIC_NAME				"Biker"
	CHARACTERISTIC_GENDER				"male"
	CHARACTERISTIC_ATTACK_SKILL			0.5
	CHARACTERISTIC_WEAPONWEIGHTS			"bots/biker_w.c"
	CHARACTERISTIC_AIM_SKILL			0.5
	CHARACTERISTIC_AIM_ACCURACY			0.5
	CHARACTERISTIC_VIEW_FACTOR			0.5
	CHARACTERISTIC_VIEW_MAXCHANGE			180
	CHARACTERISTIC_REACTIONTIME			2.5

	CHARACTERISTIC_CHAT_FILE			"bots/biker_t.c"
	CHARACTERISTIC_CHAT_NAME			"biker"
	CHARACTERISTIC_CHAT_CPM				400
	CHARACTERISTIC_CHAT_INSULT			0.95
	CHARACTERISTIC_CHAT_MISC			0.5
	CHARACTERISTIC_CHAT_STARTENDLEVEL		0.5
	CHARACTERISTIC_CHAT_ENTEREXITGAME		0.5
	CHARACTERISTIC_CHAT_KILL			0.5
	CHARACTERISTIC_CHAT_DEATH			0.5
	CHARACTERISTIC_CHAT_ENEMYSUICIDE		0.5
	CHARACTERISTIC_CHAT_HITTALKING			0.15
	CHARACTERISTIC_CHAT_HITNODEATH			0.5
	CHARACTERISTIC_CHAT_HITNOKILL			0.5
	CHARACTERISTIC_CHAT_RANDOM			0.5
	CHARACTERISTIC_CHAT_REPLY			0.175

	CHARACTERISTIC_CROUCHER				0.5
	CHARACTERISTIC_JUMPER				0.5
	CHARACTERISTIC_WEAPONJUMPING			0.5
	CHARACTERISTIC_GRAPPLE_USER			0.5

	CHARACTERISTIC_ITEMWEIGHTS			"bots/biker_i.c"
	CHARACTERISTIC_AGGRESSION			0.5
	CHARACTERISTIC_SELFPRESERVATION			0.5
	CHARACTERISTIC_VENGEFULNESS			0.5
	CHARACTERISTIC_CAMPER				0.5

	CHARACTERISTIC_EASY_FRAGGER			0.5
	CHARACTERISTIC_ALERTNESS			0.5
}

skill 5
{
	CHARACTERISTIC_NAME				"Biker"
	CHARACTERISTIC_GENDER				"male"
	CHARACTERISTIC_ATTACK_SKILL			0.95
	CHARACTERISTIC_WEAPONWEIGHTS			"bots/biker_w.c"
	CHARACTERISTIC_AIM_SKILL			1.0
	CHARACTERISTIC_AIM_ACCURACY			1.0
	CHARACTERISTIC_VIEW_FACTOR			1.0
	CHARACTERISTIC_VIEW_MAXCHANGE			240
	CHARACTERISTIC_REACTIONTIME			0.15

	CHARACTERISTIC_CHAT_FILE			"bots/biker_t.c"
	CHARACTERISTIC_CHAT_NAME			"biker"
```

```
	CHARACTERISTIC_CHAT_CPM				400
	CHARACTERISTIC_CHAT_INSULT			0.95
	CHARACTERISTIC_CHAT_MISC			0.5
	CHARACTERISTIC_CHAT_STARTENDLEVEL		0.5
	CHARACTERISTIC_CHAT_ENTEREXITGAME		0.5
	CHARACTERISTIC_CHAT_KILL			0.5
	CHARACTERISTIC_CHAT_DEATH			0.5
	CHARACTERISTIC_CHAT_ENEMYSUICIDE		0.5
	CHARACTERISTIC_CHAT_HITTALKING			0.15
	CHARACTERISTIC_CHAT_HITNODEATH			0.5
	CHARACTERISTIC_CHAT_HITNOKILL			0.5
	CHARACTERISTIC_CHAT_RANDOM			0.5
	CHARACTERISTIC_CHAT_REPLY			0.1

	CHARACTERISTIC_CROUCHER				0.0
	CHARACTERISTIC_JUMPER				1.0
	CHARACTERISTIC_WEAPONJUMPING			0.5
	CHARACTERISTIC_GRAPPLE_USER			0.5

	CHARACTERISTIC_ITEMWEIGHTS			"bots/biker_i.c"
	CHARACTERISTIC_AGGRESSION			0.5
	CHARACTERISTIC_SELFPRESERVATION			0.5
	CHARACTERISTIC_VENGEFULNESS			0.5
	CHARACTERISTIC_CAMPER				0.5

	CHARACTERISTIC_EASY_FRAGGER			0.5
	CHARACTERISTIC_ALERTNESS			0.5
}
```

```
//===========================================================================
// ***ORIGINAL CODE***
// Name:                visor_i.c
// Function:
// Programmer:          Mr Elusive (MrElusive@idsoftware.com)
// Last update:         1999-09-08
// Tab Size:            4 (real tabs)
//===========================================================================

#include "inv.h"

//initial health/armor states
#define FS_HEALTH                                       1
#define FS_ARMOR                                        1

//initial weapon weights
#define W_SHOTGUN                                       50
#define W_MACHINEGUN                                    40
#define W_GRENADELAUNCHER                               40
#define W_ROCKETLAUNCHER                                120
#define W_RAILGUN                                       85
#define W_BFG10K                                        30
#define W_LIGHTNING                                     50
#define W_PLASMAGUN                                     500

//the bot has the weapons, so the weights change a little bit
#define GWW_SHOTGUN                                     35
#define GWW_MACHINEGUN                                  50
#define GWW_GRENADELAUNCHER                             30
#define GWW_ROCKETLAUNCHER                              90
#define GWW_RAILGUN                                     25
#define GWW_BFG10K                                      41
#define GWW_LIGHTNING                                   40
#define GWW_PLASMAGUN                                   40

//initial powerup weights
#define W_TELEPORTER                                    40
#define W_MEDKIT                                        40
#define W_QUAD                                          40
#define W_ENVIRO                                        40
#define W_HASTE                                         40
#define W_INVISIBILITY                                  40
#define W_REGEN                                         40
#define W_FLIGHT                                        40

//flag weight
#define FLAG_WEIGHT                                     50

//
#include "fw_items.c"
```

```
//===========================================================================
//
// Name:                Biker_c.c
// Function:            chat lines for Biker
// Programmer:          MrElusive (MrElusive@idsoftware.com)
// Author:              The Seven Swords & R.A. Salvatore
// Editor:              Paul Jaquays
// Last update:         Oct. 10, 1999
// Tab Size:            3 (real tabs)
//===========================================================================

//example initial chats
chat "biker"
{
      //the teamplay.h file is included for all kinds of teamplay chats
      #include "teamplay.h"

      //========================================================
      //========================================================

      type "game_enter" //initiated when the bot enters the game
      {
            HELLO5;
            "Make this a good day, cuz it'll be your last.";
            "Everyone out of the pool.";
            "You just keep your hands off my hawg, ", 1, ".
Understand?";
            1, ", I'm gonna kick your sorry butt into the next
county.";
            "I squash cockroaches bigger and scarier than you, ", 1,
".";
            "This better not make me late for lunch.";
            // 0 = bot name
      } //end type

      type "game_exit" //initiated when the bot exits the game
      {
            "I have some boredom to catch up on.";
            "Screw this.";
            GOODBYE1;
            // 0 = bot name
      } //end type

      type "level_start" //initiated when a new level starts
      {
            LEVEL_START1;
            "I'm makin' ", 4, " my playground. Get off the swings, ",
fighter, "s.";
            "Where'd I park my hawg?";
            // 0 = bot name
            // 1 = randomly chosen player
            // 4 = Level's title
            } //end type

      type "level_end" //initiated when a level ends and the bot is not
first and not last in the rankings
      {
```

```
            "Waste of time.";
            LEVEL_END0;
            2, " cheats and ", 3, " sucks. Now that we know, let's go
again.";
            // 0 = bot name
            // 2 = opponent in first place
            // 3 = opponent in last place
      } //end type

      type "level_end_victory" //initiated when a level ends and the
bot is first in the rankings
      {
            "Yeehaa! Kick some doggie butt!";
            "Did that hurt ya, ", 3, "?  Bet it did!";
            LEVEL_END_VICTORY3;
            // 0 = bot name
            // 3 = opponent in last place
      } //end type

      type "level_end_lose" //initiated when a level ends and the bot
is last in the rankings
      {
            "Aw, you can kiss my bike wheel ", 2, " .... but lemme get
it spinnin' real fast first.";
            "That's it! I'm putting me a rifle rack on my hawg.";

            LEVEL_END_LOSE1;
            // 0 = bot name
      } //end type

      //=======================================================
      //=======================================================

      type "hit_talking" //bot is hit while chat balloon is visible;
lecture attacker on poor sportsmanship
      {
            //"You just became road-kill, ", fighter, ".";
            0, ", you just became a kick-stand.";
            "Nice ~one, ", 0, ". But now it's my turn.";
            //0 = shooter
      } //end type

      type "hit_nodeath" //bot is hit by an opponent's weapon attack;
either praise or insult
      {
            "C'mon, dawg, I still gots ~one good eye!";
            "Don't dirty the leathers, ", 0,"!";
            "You cow-poker! I'll show you what's what!";
            "OK, you gave it your best and now it's my turn.";
            "I'm gonna hurt you, ", 0, ". I'm gonna hurt you bad.";
            DEATH_INSULT2;
            //0 = shooter
      } //end type

      type "hit_nokill" //bot hits an opponent but does not kill it
      {
            TAUNT1;
```

```
            "Yer as slow as ya are ugly.";
            TAUNT8;
            "You look mighty pretty in blood red.";
            "Lousy, cheap ammo.";
            "Sheee-oooot! I gotta get me a bigger gun than this ~one.";
            //0 = opponent
      } //end type

      //=======================================================
      //=======================================================

      type "death_telefrag" //initiated when the bot is killed by a
telefrag
      {
            "Them trucks never give a hawg-rider no respect.";
            "That was some serious road rash.";
            "Anybody get the license plate on that elephant?";
            DEATH_TELEFRAGGED0;
            // 0 = enemy name
      } //end type

      type "death_cratered" //initiated when the bot is killed by
falling damage
      {
            "Them Kneivel boys ain't got nothin' on me.";
            "May be somethin' to them helmet laws ...";
            "Dang! Colorado Rocky Mountain high!";
            // 0 = enemy name
      } //end type

      type "death_lava" //initiated when the bot dies in lava
      {
            "Dang! That's hotter than a tailpipe runnin' jet fuel.";
            "Hoooooweeee! Too much lighter fluid on the grill there.";
            DEATH_LAVA1;
            // 0 = enemy name
      } //end type

      type "death_slime" //initiated when the bot dies in slime
      {
            "Any biscuits with this gravy?";
            "This is like showerin' in degreaser.";
            "Hey Lucy, this stuff's better'n yer maw's cookin'!";
            // 0 = enemy name
      } //end type

      type "death_drown" //initiated when the bot drowns
      {
            "Leather's heavy when it's wet. Dang.";
            "Great, now I smells like a wet dog.";
            "Stupid place for a swimmmin' pool.";
            // 0 = enemy name
      } //end type

      type "death_suicide" //initiated when bot blows self up with a
weapon or craters
      {
```

```
            "You so much as giggle, ", 0, ", and I break yer face.";
            "Did that truck run over my bike too?";
            DEATH_SUICIDE0;
            DEATH_SUICIDE5;
            // 0 = enemy name
      } //end type

      type "death_gauntlet" //initiated when the bot is killed by a
gauntlet attack
      {
            "Get yer slimy paws off o' me, ", 0, "!";
            "Last guy that touched me can only count to ~five now.";
            "Ain't nobody touches me like that.";
            DEATH_GAUNTLET2;
            // 0 = enemy name
      } //end type

      type "death_rail" //initiated when the bot is killed by a rail
gun shot
      {
            DEATH_RAIL2;
            "This ain't no campground, ", 0, ".";
            "Let's settle this outside, face to face.";
            // 0 = enemy name
      } //end type

      type "death_bfg" //initiated when the bot died by a BFG
      {
            "And the friggin' rockets red ... uhhh ... green glare!";
            DEATH_BFG2;
            "Ha, ya missed, ", 0, "! 'Almost' only counts in hand
grenades and ... oh, puke ...";
            "Stampede!";
            // 0 = enemy name
      } //end type

      type "death_insult" //insult initiated when the bot died
      {
            "I take my coffee stronger'n you, ", 0, ".";
            "You just stepped on the highway to hell, ", fighter, ".";
            "I can't believe a ", fighter, " like ", 0, " fragged me!";
            DEATH_INSULT0;
            DEATH_INSULT4;
            DEATH_INSULT5;
            // 0 = enemy name
      } //end type

      type "death_praise" //praise initiated when the bot died
      {
            "Yeah, big hairy deal, ", 0, ". Ya killed me. So what?";
            "Well, ain't ", 0, " a ~hero?  Come and get your special
prize.";
            D_PRAISE2;
            0, " killed me to death! Dang!";
            // 0 = enemy name
      } //end type
```

```
	//=====================================================
	//=====================================================

	type "kill_rail" //initiated when the bot kills someone with rail
gun
	{
		"Turn into the wind, ", fighter, ", I wanna hear ya
whistle.";
		KILL_RAIL1;
		KILL_RAIL0;
		// 0 = enemy name
	} //end type

	type "kill_gauntlet" //initiated when the bot kills someone with
gauntlet
	{
		"Up close and impersonal.";
		"Like swattin' flies offa roadkill.";
		"That's the kind of action I like most.";
		"Yeah! Gimme some more of that!";
		KILL_GAUNTLET0;
		// 0 = enemy name
	} //end type

	type "kill_telefrag" //initiated when the bot telefragged someone
	{
		"Here's the beef!";
		"Get off the Road!";
		"Popped you like a Prairie Oyster!";
		TELEFRAGGED0;
		// 0 = enemy name
	} //end type

	type "kill_suicide" //initiated when the player kills self
	{
		"Hey moron, AIM, then fire!";
		"Must tickle, cuz I'm laughin'";
		// 0 = enemy name
	} //end type

	type "kill_insult" //Insult initiated when the bot killed someone
	{
		"I'm gonna bury ", 0, " face down so I got a place to park
my bike.";
		"Ahhhh, you ain't worth the trouble to run you over.";
		"Biker 1, Roadkill 0.";
		KILL_INSULT0;
		KILL_INSULT4;
		KILL_INSULT8;
		// 0 = enemy name
	} //end type

	type "kill_praise" //praise initiated when the bot killed someone
	{
		KILL_INSULT36;
		"You'll make good maggot food, ", 0, ". That's something.";
		"You was brain-dead anyway, ", 0, ".";
```

```
            // 0 = enemy name
      } //end type

      //=========================================================
      //=========================================================

      type "random_insult" //insult initiated randomly (just when the
bot feels like it)
      {
            TAUNT7;
            MISC6;
            "All the sex appeal of open heart surgery!";
            "Yer like sand on the road, ", 0, ".I just blow it away.";
            "Your mamma's gonna see you on a milk carton, ", 0, ".";
            "You stay away from my dawg, ", 0, ".";
            // 0 = name of randomly chosen player
            // 1 = bot name
      } //end type

      type "random_misc" //miscellanous chats initiated randomly
      {
            GUYTALK2;
            "I eat my road kill.";
            "*[Bu-u-u-u-u-u-urp]* ";
            "Anybody got a ~six-pack on 'em?";
            "Okay, any o' you pretty ladies want a ride on my hawg?";
            one_liners;
            "Don't go tellin' Lucy I was here ... she'll whip my
butt.";
            femalebot, " better git her ass back in the kitchen and
finish making me that chicken pot pie!";
            MISC0;
            MISC8;
            MISC12;
            // 0 = name of randomly chosen player
            // 1 = bot name
      } //end type
} //end biker chat
```

```
//======================================================================
//
// Name:                visor_w.c
// Function:
// Programmer:
// Last update:
// Tab Size:            4 (real tabs)
//======================================================================

#include "inv.h"

#define W_GAUNTLET                                      10
#define W_SHOTGUN                                       30
#define W_MACHINEGUN                                    20
#define W_GRENADELAUNCHER                               40
#define W_ROCKETLAUNCHER                                100
#define W_RAILGUN                                       70
#define W_BFG10K                                        95
#define W_LIGHTNING                                     80
#define W_PLASMAGUN                                     275
#define W_GRAPPLE                                       15

//
#include "fw_weap.c"
```

APPENDIX C. Q3A MENU CODE

```
//
/*
=======================================================
RATE PLAYER MENU

Skip Morrow
Jeff DeBrine
July 2000

Other Q3A menu files were used as a template
Contains original code for new menu choices
=======================================================
*/

#include "ui_local.h"

#define ART_FRAME       "menu/art/addbotframe"
#define ART_BACK0       "menu/art/back_0"
#define ART_BACK1       "menu/art/back_1"

#define RATE_PLAYER_MENU_VERTICAL_SPACING   28

#define ID_EXCELLENT      10
#define ID_AVERAGE        11
#define ID_BELOW_AVERAGE  12
#define ID_POOR           13

typedef struct {
    menuframework_s menu;

    menutext_s      banner;
    menutext_s      excellent;
    menutext_s      average;
    menutext_s      belowAverage;
    menutext_s      poor;
    menubitmap_s    frame;
    menulist_s      list;
    menubitmap_s    back;

    int             gametype;
    int             numBots;
    int             selectedBot;
    char            *bots[9];
    char            botNames[9][16];
} ratePlayerMenuInfo_t;

static ratePlayerMenuInfo_t ratePlayerMenuInfo;

#define NUM_RATINGS     3
static const char *ratings[] = {
    "Excellent",
    "Average",
    "Below Average",
    "Poor",
```

```
    NULL
};

/*
===============
UI_RatePlayerMenu_BackEvent
===============
*/
static void UI_RatePlayerMenu_BackEvent( void *ptr, int event ) {
    if( event != QM_ACTIVATED ) {
        return;
    }
    UI_PopMenu();
}

/*
===============
UI_RatePlayerMenu_SetList
===============
*/
static void UI_RatePlayerMenu_SetList( int id ) {
    ratePlayerMenuInfo.list.generic.id = id;
    ratePlayerMenuInfo.list.numitems = NUM_RATINGS;
    ratePlayerMenuInfo.list.itemnames = ratings;
    ratePlayerMenuInfo.list.generic.bottom =
    ratePlayerMenuInfo.list.generic.top +
    ratePlayerMenuInfo.list.numitems * PROP_HEIGHT;
}

/*
=================
RatePlayer_Event
=================
*/
void RatePlayer_Event( void *ptr, int notification ) {
    if( notification != QM_ACTIVATED ) {
        return;
    }

    switch( ((menucommon_s*)ptr)->id ) {
    case ID_EXCELLENT:
        UI_PopMenu();
        break;

    case ID_AVERAGE:
        UI_PopMenu();
        break;

    case ID_BELOW_AVERAGE:
        UI_PopMenu();
        break;

    case ID_POOR:
        UI_PopMenu();
        break;
    }
}
```

```
/*
=================
UI_RatePlayerMenu_Key
=================
*/
sfxHandle_t UI_RatePlayerMenu_Key( int key ) {
    menulist_s  *l;
    int x;
    int y;
    int index;

    l = (menulist_s *)Menu_ItemAtCursor( &ratePlayerMenuInfo.menu );
    if( l != &ratePlayerMenuInfo.list ) {
        return Menu_DefaultKey( &ratePlayerMenuInfo.menu, key );
    }

    switch( key ) {
        case K_MOUSE1:
            x = l->generic.left;
            y = l->generic.top;
            if( UI_CursorInRect( x, y, l->generic.right - x, l-
                                                >generic.bottom - y ) ) {
                index = (uis.cursory - y) / PROP_HEIGHT;
                l->oldvalue = l->curvalue;
                l->curvalue = index;

                if( l->generic.callback ) {
                    l->generic.callback( l, QM_ACTIVATED );
                    return menu_move_sound;
                }
            }
            return menu_null_sound;

        case K_KP_UPARROW:
        case K_UPARROW:
            l->oldvalue = l->curvalue;

            if( l->curvalue == 0 ) {
                l->curvalue = l->numitems - 1;
            }
            else {
                l->curvalue--;
            }
            return menu_move_sound;

        case K_KP_DOWNARROW:
        case K_DOWNARROW:
            l->oldvalue = l->curvalue;

            if( l->curvalue == l->numitems - 1 ) {
                l->curvalue = 0;;
            }
            else {
                l->curvalue++;
            }
            return menu_move_sound;
```

```
    }

    return Menu_DefaultKey( &ratePlayerMenuInfo.menu, key );
}

/*
================
UI_RatePlayerMenu_ListDraw
================
*/
static void UI_RatePlayerMenu_ListDraw( void *self ) {
    menulist_s  *l;
    int         x;
    int         y;
    int         i;
    float       *color;
    qboolean    hasfocus;
    int         style;

    l = (menulist_s *)self;

    hasfocus = (l->generic.parent->cursor == l->generic.menuPosition);

    x = 320;//l->generic.x;
    y = l->generic.y;
    for( i = 0; i < l->numitems; i++ ) {
        style = UI_LEFT|UI_SMALLFONT|UI_CENTER;
        if( i == l->curvalue ) {
            color = color_yellow;
            if( hasfocus ) {
                style |= UI_PULSE;
            }
        }
        else {
            color = color_orange;
        }

        UI_DrawProportionalString( x, y, l->itemnames[i], style, color
);
        y += PROP_HEIGHT;
    }
}

/*
===============
UI_RatePlayerMenu_ListEvent
===============
*/
static void UI_RatePlayerMenu_ListEvent( void *ptr, int event ) {
    int     id;
    int     selection;

    if (event != QM_ACTIVATED)
        return;

    id = ((menulist_s *)ptr)->generic.id;
    selection = ((menulist_s *)ptr)->curvalue;
```

```
    UI_PopMenu();
}

/*
===============
UI_RatePlayerMenu_BuildBotList
===============
*/
static void UI_RatePlayerMenu_BuildBotList( void ) {
    uiClientState_t cs;
    int     numPlayers;
    int     isBot;
    int     n;
    char    playerTeam;
    char    botTeam;
    char    info[MAX_INFO_STRING];

    for( n = 0; n < 9; n++ ) {
        ratePlayerMenuInfo.bots[n] = ratePlayerMenuInfo.botNames[n];
    }

    trap_GetClientState( &cs );

    Q_strncpyz( ratePlayerMenuInfo.botNames[0], "Everyone", 16 );
    ratePlayerMenuInfo.numBots = 1;

    trap_GetConfigString( CS_SERVERINFO, info, sizeof(info) );
    numPlayers = atoi( Info_ValueForKey( info, "sv_maxclients" ) );
    ratePlayerMenuInfo.gametype = atoi( Info_ValueForKey( info,
                                                    "g_gametype" ) );

    for( n = 0; n < numPlayers && ratePlayerMenuInfo.numBots < 9; n++ )
{
        trap_GetConfigString( CS_PLAYERS + n, info, MAX_INFO_STRING );

        if( n == cs.clientNum ) {
            playerTeam = *Info_ValueForKey( info, "t" );
            continue;
        }

        isBot = atoi( Info_ValueForKey( info, "skill" ) );
        if( !isBot ) {
            continue;
        }

        botTeam = *Info_ValueForKey( info, "t" );
        if( botTeam != playerTeam ) {
            continue;
        }

        Q_strncpyz(
           ratePlayerMenuInfo.botNames[ratePlayerMenuInfo.numBots],
           Info_ValueForKey( info, "n" ), 16 );
        Q_CleanStr(
           ratePlayerMenuInfo.botNames[ratePlayerMenuInfo.numBots] );
           ratePlayerMenuInfo.numBots++;
```

```
    }
}

/*
===============
UI_RatePlayerMenu_Init
===============
*/
static void UI_RatePlayerMenu_Init( void ) {
    int     y;

    UI_RatePlayerMenu_Cache();

    trap_R_RegisterShaderNoMip( ART_FRAME );
    trap_R_RegisterShaderNoMip( ART_BACK0 );
    trap_R_RegisterShaderNoMip( ART_BACK1 );

    memset( &ratePlayerMenuInfo, 0, sizeof(ratePlayerMenuInfo) );
    ratePlayerMenuInfo.menu.fullscreen = qfalse;
    ratePlayerMenuInfo.menu.key = UI_RatePlayerMenu_Key;

    UI_RatePlayerMenu_BuildBotList();

    ratePlayerMenuInfo.banner.generic.type      =          MTYPE_BTEXT;
    ratePlayerMenuInfo.banner.generic.x         =                  320;
    ratePlayerMenuInfo.banner.generic.y         =                   16;
    ratePlayerMenuInfo.banner.string            =        "RATE PLAYER";
    ratePlayerMenuInfo.banner.color             =          color_white;
    ratePlayerMenuInfo.banner.style             =            UI_CENTER;

    ratePlayerMenuInfo.frame.generic.type       =         MTYPE_BITMAP;
    ratePlayerMenuInfo.frame.generic.flags      =         QMF_INACTIVE;
    ratePlayerMenuInfo.frame.generic.name       =            ART_FRAME;
    ratePlayerMenuInfo.frame.generic.x          =              320-150;
    ratePlayerMenuInfo.frame.generic.y          =              240-100;
    ratePlayerMenuInfo.frame.width              =            300; //466
    ratePlayerMenuInfo.frame.height             =            160; //332

    ratePlayerMenuInfo.list.generic.type        =     MTYPE_SCROLLLIST;
    ratePlayerMenuInfo.list.generic.flags       =     QMF_PULSEIFFOCUS;
    ratePlayerMenuInfo.list.generic.ownerdraw   =
                                          UI_RatePlayerMenu_ListDraw;
    ratePlayerMenuInfo.list.generic.callback    =
                                         UI_RatePlayerMenu_ListEvent;
    ratePlayerMenuInfo.list.generic.x           =               320-64;
    ratePlayerMenuInfo.list.generic.y           =                  120;

    y = 170;
    ratePlayerMenuInfo.excellent.generic.type           =  MTYPE_PTEXT;
    ratePlayerMenuInfo.excellent.generic.flags          =
                                 QMF_CENTER_JUSTIFY|QMF_PULSEIFFOCUS;
    ratePlayerMenuInfo.excellent.generic.x              =          320;
    ratePlayerMenuInfo.excellent.generic.y              =            y;
    ratePlayerMenuInfo.excellent.generic.id             = ID_EXCELLENT;
    ratePlayerMenuInfo.excellent.generic.callback       =
                                                       RatePlayer_Event;
    ratePlayerMenuInfo.excellent.string                 =  "EXCELLENT";
```

```
ratePlayerMenuInfo.excellent.color                  =    color_red;

ratePlayerMenuInfo.excellent.style                  =
                                              UI_CENTER|UI_SMALLFONT;

y += RATE_PLAYER_MENU_VERTICAL_SPACING;
ratePlayerMenuInfo.average.generic.type             = MTYPE_PTEXT;
ratePlayerMenuInfo.average.generic.flags            =
                               QMF_CENTER_JUSTIFY|QMF_PULSEIFFOCUS;
ratePlayerMenuInfo.average.generic.x                =         320;
ratePlayerMenuInfo.average.generic.y                =           y;
ratePlayerMenuInfo.average.generic.id               =  ID_AVERAGE;
ratePlayerMenuInfo.average.generic.callback         =
                                                   RatePlayer_Event;
ratePlayerMenuInfo.average.string                   =   "AVERAGE";
ratePlayerMenuInfo.average.color                    =   color_red;
ratePlayerMenuInfo.average.style                    =
                                              UI_CENTER|UI_SMALLFONT;

y += RATE_PLAYER_MENU_VERTICAL_SPACING;
ratePlayerMenuInfo.belowAverage.generic.type        =
                                                     MTYPE_PTEXT;
ratePlayerMenuInfo.belowAverage.generic.flags       =
                               QMF_CENTER_JUSTIFY|QMF_PULSEIFFOCUS;
ratePlayerMenuInfo.belowAverage.generic.x           =         320;
ratePlayerMenuInfo.belowAverage.generic.y           =           y;
ratePlayerMenuInfo.belowAverage.generic.id          =
                                                   ID_BELOW_AVERAGE;
ratePlayerMenuInfo.belowAverage.generic.callback    =
                                                   RatePlayer_Event;
ratePlayerMenuInfo.belowAverage.string              =
                                                    "BELOW AVERAGE";
ratePlayerMenuInfo.belowAverage.color               =   color_red;
      ratePlayerMenuInfo.belowAverage.style         =
                                              UI_CENTER|UI_SMALLFONT;

y += RATE_PLAYER_MENU_VERTICAL_SPACING;
ratePlayerMenuInfo.poor.generic.type                =
                                                     MTYPE_PTEXT;
      ratePlayerMenuInfo.poor.generic.flags         =

  QMF_CENTER_JUSTIFY|QMF_PULSEIFFOCUS;
ratePlayerMenuInfo.poor.generic.x                   =         320;
ratePlayerMenuInfo.poor.generic.y                   =           y;
ratePlayerMenuInfo.poor.generic.id                  =     ID_POOR;
ratePlayerMenuInfo.poor.generic.callback        =

                                                   RatePlayer_Event;

ratePlayerMenuInfo.poor.string                      =      "POOR";
ratePlayerMenuInfo.poor.color                       =   color_red;
ratePlayerMenuInfo.poor.style                       =
                                              UI_CENTER|UI_SMALLFONT;

ratePlayerMenuInfo.back.generic.type                =MTYPE_BITMAP;
ratePlayerMenuInfo.back.generic.name                =   ART_BACK0;
```

```
    ratePlayerMenuInfo.back.generic.flags              =
                                  QMF_LEFT_JUSTIFY|QMF_PULSEIFFOCUS;
    ratePlayerMenuInfo.back.generic.callback           =
                                       UI_RatePlayerMenu_BackEvent;
    ratePlayerMenuInfo.back.generic.x                  =           0;
    ratePlayerMenuInfo.back.generic.y                  =      480-64;
    ratePlayerMenuInfo.back.width                      =         128;
    ratePlayerMenuInfo.back.height                     =          64;
    ratePlayerMenuInfo.back.focuspic                   =   ART_BACK1;

    Menu_AddItem( &ratePlayerMenuInfo.menu, &ratePlayerMenuInfo.banner
                                                                    );
    Menu_AddItem( &ratePlayerMenuInfo.menu, &ratePlayerMenuInfo.frame
                                                                    );
    Menu_AddItem( &ratePlayerMenuInfo.menu, &ratePlayerMenuInfo.list );
    Menu_AddItem( &ratePlayerMenuInfo.menu, &ratePlayerMenuInfo.back );
    Menu_AddItem( &ratePlayerMenuInfo.menu,
                                     &ratePlayerMenuInfo.excellent );
    Menu_AddItem( &ratePlayerMenuInfo.menu, &ratePlayerMenuInfo.average
                                                                    );
    Menu_AddItem( &ratePlayerMenuInfo.menu,
                                  &ratePlayerMenuInfo.belowAverage );
    Menu_AddItem( &ratePlayerMenuInfo.menu, &ratePlayerMenuInfo.poor );

    ratePlayerMenuInfo.list.generic.left               =         220;
    ratePlayerMenuInfo.list.generic.top                =
                                   ratePlayerMenuInfo.list.generic.y;
    ratePlayerMenuInfo.list.generic.right              =         420;
//  UI_RatePlayerMenu_SetList( ID_LIST_BOTS );
}

/*
=================
UI_RatePlayerMenu_Cache
=================
*/
void UI_RatePlayerMenu_Cache( void ) {
    trap_R_RegisterShaderNoMip( ART_FRAME );
    trap_R_RegisterShaderNoMip( ART_BACK0 );
    trap_R_RegisterShaderNoMip( ART_BACK1 );
}

/*
===============
UI_RatePlayerMenu
===============
*/
void UI_RatePlayerMenu( void ) {
    UI_RatePlayerMenu_Init();
    UI_PushMenu( &ratePlayerMenuInfo.menu );
}

/*
===============
UI_RatePlayerMenu_f
===============
*/
```

```
void UI_RatePlayerMenu_f( void ) {
    uiClientState_t cs;
    char    info[MAX_INFO_STRING];
    int     team;

    // make sure it's a team game
    trap_GetConfigString( CS_SERVERINFO, info, sizeof(info) );
    ratePlayerMenuInfo.gametype = atoi( Info_ValueForKey( info,
                                                    "g_gametype" ) );
    if( ratePlayerMenuInfo.gametype < GT_TEAM ) {
        return;
    }

    // not available to spectators
    trap_GetClientState( &cs );
    trap_GetConfigString( CS_PLAYERS + cs.clientNum, info,
                                                    MAX_INFO_STRING );
    team = atoi( Info_ValueForKey( info, "t" ) );
    if( team == TEAM_SPECTATOR ) {
        return;
    }

    UI_RatePlayerMenu();
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Forsyth, D. (1990). Group Structure. Group Dynamics. Retrieved April 21, 2000 from the World Wide Web: http://www.vcu.edu/hasweb/psy/psy633/read.html#amer

Foushee, H. (1984). Dyads and triads at 35,000 feet: factors affecting group process and aircrew performance [Abstract]. American Psychologist, 39, 885-893. Retrieved April 21, 2000 from the World Wide Web: http://www.vcu.edu/hasweb/psy/psy633/foushee.htm

Knight, C., & Munro, M. (1998). Using an existing game engine to facilitate multi-user information visualization. Retrieved May 15, 2000 from the World Wide Web: http://www.dur.ac.uk/~dcs1crk/workfiles/documents/multi-user_software_vis/multi-user_software_vis.html

McCraw, R., & Bearden, D. (1990). Personality factors in failure to adapt to the military [Abstract]. Military Medicine, 155(3), 127-130. Retrieved April 24, 2000 from the PsycINFO database on the World Wide Web: http://psycinfo.apa.org

McDonald, B. (1999). Bot Weapon/Itemweights Editing Guide. Retrieved August 18, 2000 from the World Wide Web: http://www.planetquake.com/quake3/q3aguide/bot-tutorial_intro.shtml

Miliano, V. (1999). Application of a 3D game engine to enhance the design, visualization and presentation of commercial real estate. Retrieved June 23, 2000 from the World Wide Web: http://www.unrealty.net/vsmm99/

National Research Council. (1997). Modeling and Simulation: Linking Entertainment and Defense. Washington D.C.: National Academy Press.

Singhal, S., & Zyda, M. (1999). Networked Virtual Environments: Design and Implementation. New York: Addison-Wesley Company, Inc.

Vince, J. (1992). 3-D Computer Animation. New York: Addison-Wesley Publishing Company Inc.

Weiner, H. (1990). Group-level and individual-level mediators of the relationship between soldier satisfaction with social support and performance motivation [Abstract]. Military Psychology, 2(1), 21-32. Retrieved April 24, 2000 from PsycINFO database on the World Wide Web: http://psycinfo.apa.org

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. RADM Richard Mayo 1
   CNO, N6
   2000 Navy Pentagon
   Washington, DC 20350-2000

3. CDR George Phillips, USN (Ret) 1
   CNO, N6M1
   2000 Navy Pentagon
   Washington, DC 20350-2000

4. Dr. Allen Zeman 1
   CNO, N7B
   2000 Navy Pentagon
   Washington, DC 20350-2000

5. Dr. Michael Macedonia 1
   Chief Scientist and Technical Director
   US Army STRICOM
   12350 Research Parkway
   Orlando, FL 32826-3276

6. Dr. Casey Wardynski 2
   United States Military Academy
   Office of Economic & Manpower Analysis
   607 Cullum Rd, Floor 1B, Rm B109
   West Point, NY 10996-1798

7. Dr. Michael Capps 3
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

8. LT Jeffrey D. DeBrine 1
   925 Langdon Ct.
   Annapolis, MD 21403

9. LT Donald E. Morrow C/O Mike Lewis 1
   115 E. Williams St.
   Broadwell, IL 62623